Taming Complexity of Large Software Systems: Contracting, Self-Adaptation and Feature Modeling

Philippe Collet

UNIVERSITÉ DE NICE–SOPHIA ANTIPOLIS — UFR Sciences
École Doctorale de Sciences et Technologies de l'Information
et de la Communication (STIC)

HABILITATION A DIRIGER DES RECHERCHES

de l'UNIVERSITÉ de Nice–Sophia Antipolis

Discipline: Informatique

présentée et soutenue publiquement par

Philippe COLLET

Taming Complexity of Large Software Systems: Contracting, Self-Adaptation and Feature Modeling

soutenue le 6 décembre 2011 devant le jury composé de:

Président	Jean-Claude BERMOND	Directeur de Recherche, CNRS I3S - Sophia Antipolis
Rapporteurs	Betty H.C. CHENG	Professor, Michigan State University (USA)
	Ivica CRNKOVIC	Professor, Mälardalen University (Sweden)
	Patrick HEYMANS	Professeur, Université de Namur FUNDP (Belgique)
Examinateurs	Laurence DUCHIEN	Professeur, Université de Lille 1
	Philippe LAHIRE	Professeur, Université Nice - Sophia Antipolis
	Jacques MALENFANT	Professeur, Université Pierre et Marie Curie, Paris

Abstract

Our work stands in the field of software engineering for large scale software intensive systems. We aim at providing techniques and tools to help software architects master the ever-increasing complexity of these systems. Using mainly model-driven engineering approaches, our contribution is organised around three axes. The first axis concerns the development of reliable and flexible hierarchical component-based systems with dynamic reconfiguration capabilities. Through the use of novel forms of software contracts, the proposed systems and frameworks support several specification formalisms and maintain up-to-date contracts at runtime. A second part of our work consists in providing self-adaptive capabilities to these contracting systems, through negotiation mechanisms over contracts and self-adaptive monitoring sub-systems. A third axis is related to software product lines in which feature models are widely used to model variability. Our contribution consists in providing a set of sound and efficiently implemented composition operators for feature models, as well as a dedicated language for their large scale management.

Résumé

Nos travaux s'inscrivent dans le domaine du génie logiciel pour les systèmes informatiques à large échelle. Notre objectif est de fournir des techniques et des outils pour aider les architectes logiciels à maîtriser la complexité toujours grandissante de ces systèmes. Principalement fondées sur des approches par ingénierie des modèles, nos contributions s'organisent autour de trois axes. Le premier axe concerne le développement de systèmes à la fois fiables et flexibles, et ce à base de composants hiérarchiques équipés de capacités de reconfiguration dynamique. Par l'utilisation de nouvelles formes de contrats logiciels, les systèmes et *frameworks* que nous proposons prennent en compte differents formalismes de spécification et maintiennent les contrats à jour pendant l'exécution. Une seconde partie de nos travaux s'intéresse à fournir des capacités auto-adaptatives à ces systèmes contractuels, à travers des mécanismes de négociation de contrats et des sous-systèmes de *monitoring* eux-mêmes auto-adaptatifs. Un troisième axe concerne les lignes de produits logiciels dans lesquelles les *features models* sont largement utilisés pour modéliser la variabilité. Nos contributions consistent en un ensemble d'opérateurs de composition bien définis et implémentés efficacement pour les *feature models*, ainsi qu'un langage dédié permettant leur gestion à large échelle.

Acknowledgments

First, I would like to thank Roger Rousseau, my former PhD advisor, who taught me software engineering when I was a master student and finally supervised my PhD, teaching me how to make research. I am still remembering quite often when, at the end of a brainstorming, he put his both hands on the back of his head and sharply summarized the issues and solutions. I now make the same moves quite often in a meeting, always thinking of Roger. I would also like to thank Philippe Lahire, with whom I have conducted a part of the presented research work. He is both a colleague and a *stainless* friend, every day we are more complementary. Continuing to make research with him is going to be both fun and fruitful. Mireille Blay-Fornarino has also an important impact on my research, being there every time you need her, exhausting duties feel less hard when working with her.

This document presents some of the work done by the OCL, Rainbow and Modalis research groups. This habilitation could not have been possible without the help of these group members. I notably thanks the following members: Robert Chignoli, Pierre Crescenzo, Alban Gaignard, Stéphane Lavirotte, Diane Lingrand, Anne-Marie Pinna-Dery, Hélène Renard, Philippe Renevier, Gaëtan Rey, Jean-Yves Tigli. Many thanks also go to Michel Riveill, who hosted me in the Rainbow team, and Johan Montagnat, who convinced me to jump in the Modalis adventure.

I would also like to thank the jury members, starting by my three reviewers, Betty H.C. Cheng, Ivica Crnkovic and Patrick Heymans. They accepted to make some room in their busy schedule to review my work. I also have to thank Laurence Duchien and Jacques Malenfant, working with them in projects, supervising students were always interesting. I also thanks Jean-Claude Bermond for accepting to chair this jury.

Part of the presented work has also been conducted in collaboration with other researchers. I would like to notably thank Thierry Coupaye and Nicolas Rivierre, as they put some trust in me through our first collaboration contract with France Télécom R&D. Working directly with Nicolas was a great pleasure. I would also like to thank Robert B. France for our, mostly distant, collaborative work on feature modeling. One day, I would like to have his capability to analyse a research problem and propose solutions. Many thanks also go to Anthony Cleve, Daniel Deveaux, Philippe Merle, Sabine Moisan, Jean-Paul Rigault and Isis Truck for our discussions and collaborative work. A special thanks goes to Xavier Blanc, who decided me to finish off this document around a plate of mussels in Lille.

The presented contributions were also realised through the work and supervisions of several PhD students: Alain Ozanne, Hervé Chang, Bao Le Duc and Mathieu Acher. Thanks to each of you, it was a great adventure to help you reach your PhD goal. A special thanks to Filip Krikava, my ongoing PhD student, who is living the adventure now.

Finally, I would like to thank all my family for tolerating the constant invasion of research, teaching and administrative duties in their life and for supporting me anyway.

It is the pervading law of all things organic and inorganic,
Of all things physical and metaphysical,
Of all things human and all things super-human,
Of all true manifestations of the head,
Of the heart, of the soul,
That the life is recognizable in its expression,
***That form ever follows function**. This is the law.*

Louis Sullivan[1]

[1] "The Tall Office Building Artistically Considered", Lippincott's Magazine (March 1896).

Contents

Chapter 1 Introduction **1**
1.1 Context and Approach . . . 1
1.2 Contracting Software . . . 3
1.3 Self-Adaptive Capabilities . . . 4
1.4 Feature Modeling in Software Product Lines . . . 5
1.5 Outline . . . 6

Chapter 2 Contracting **9**
2.1 A Contracting System for Hierarchical Components . . . 10
2.2 From a Contracting System to a Framework . . . 21
2.3 From a Framework to a Model-Driven Toolchain . . . 35
2.4 Contract-based Self-testable Components . . . 44

Chapter 3 Self-Adaptation **53**
3.1 Negotiation of Non-functional Contracts . . . 54
3.2 Compositional Patterns of Non-Functional Properties . . . 66
3.3 Self-adaptive QoI-aware Monitoring . . . 80

Chapter 4 Feature Modeling **91**
4.1 Supporting Separation of Concerns for FM Management . . . 92
4.2 A Domain-Specific Language for Large Scale Management of FM . . . 101
4.3 Applications of SoC in Feature Modeling . . . 109

Chapter 5 Conclusions **119**
5.1 Assessment . . . 119
5.2 A Research Roadmap . . . 122

Bibliography **127**

List of Figures

2.1 A multimedia player in Fractal. . . . 12
2.2 Example of interface contract. . . . 15
2.3 External composition contract on component <pl>. . . . 15
2.4 Internal composition contract on component <fp>. . . . 16
2.5 Responsibilities for an interface contract, with example of fig. 2.2. . . . 17
2.6 Responsibilities for an external composition contract, with example of fig. 2.3. . . . 17
2.7 Architecture of the server. . . . 24
2.8 Principles of the *Interact* framework. . . . 25
2.9 Concrete syntax pattern of a resulting contract object. . . . 26
2.10 Agreement of the assertion-based contract between <af> and <c>. . . . 29
2.11 Kernel model of the contracting framework. . . . 32
2.12 Roles and features of the framework with a Fractal instantiation. . . . 33
2.13 Main interactions related to the contract controller in the framework. . . . 34
2.14 Models and metamodels in the FAROS process. . . . 37
2.15 Structural part of the central metamodel (from FAROS deliverable F-2.3 [DBFC+08]). 38
2.16 Complete contract metamodel (extracted from [DBFC+08]). . . . 39
2.17 TimeoutContract on InformationProvider (extracted from [DBFC+08]). . . . 41
2.18 CCL-J metamodel (extracted from [DBFC+08]). . . . 42
2.19 Transformation of the Timer contract into *CCL-J* model (extracted from [DBFC+08]). 43
2.20 CurrencyConverter architecture. . . . 45
2.21 Resulting contracts on the CurrencyConverter. . . . 47
2.22 Black-box testing in isolation. . . . 49
2.23 Requirement testing. . . . 50
2.24 Integration testing. . . . 51

3.1 External composition contract on the *Fractal* multimedia player. . . . 56
3.2 Negotiating parties for the precondition of the external composition contract. . . . 57
3.3 Concession-based negotiation process. . . . 59
3.4 Action proposals in the effort-based negotiation policy. . . . 61
3.5 Component-based architecture of an atomic negotiation component. . . . 64
3.6 Overview of modeling patterns. . . . 68
3.7 Examples of compositional relations. . . . 70
3.8 Elements at the meta-level. . . . 70
3.9 Patterns for Fractal components. . . . 71
3.10 Architecture of the server, and some contracts. . . . 73
3.11 Overview of the propagative negotiation process. . . . 74
3.12 Propagative scheme for the `maxUsers` property. . . . 76
3.13 Propagative scheme for the `groupedUsersRatio` property. . . . 77
3.14 Propagative scheme for the `bdWidthLevel` property. . . . 77

3.15 High-level architecture of atomic negotiations in component membranes. 78
3.16 ADAMO functional architecture and roles. 83
3.17 Example of temporal filter (extracted from [LDCMR10]). 85
3.18 ADAMO self-adaptive loop. 88

4.1 FM, set of configurations and propositional logic encoding. 92
4.2 Merging in strict union and diff modes. 97
4.3 Merging in intersection mode. 97
4.4 Example of slice applied on the feature model of Figure. 4.1a. 99
4.5 FM on medical image format. 104
4.6 Two FMs $fm4$ and $fm5$ at the end of a *FAMILIAR* script execution 107
4.7 Merge: semantic properties and *FAMILIAR* notation. 107
4.8 Process overview: from design to configuration of the workflow. 110
4.9 From requirements to deployment and runtime: process. 113
4.10 Extraction and refinement process of architectural FMs. 114
4.11 *FAMILIAR* and SoC operators: case studies. 117

CHAPTER 1

Introduction

This habilitation thesis should be seen as a summary of the research conducted in the field of software engineering over a period of around ten years, it is not another PhD thesis.

This chapter presents the context of our work, summarizes our contributions according to three research axes and gives an outline of the rest of this document.

1.1 Context and Approach

By the late 60s, it was already clear that even if computer systems were made of more than just software, this software part was the most important to achieve envisioned goals of organizational improvement in many human and machine-based activities. Unfortunately the software community was also acknowledging the fact that failing software was the main impediment to these objectives. During the 1968 NATO Conference, this community establishes the software development activity as an engineering problem, defining goals for the discipline of *software engineering*. In more than forty years, software engineering has made huge progress, both in theory and practice, but the complexity of software intensive systems constantly and inexorably grew and almost annihilated the successive improvements of the discipline. Over the two last decades, this complexity led to huge costs in both distributing them to end-users and maintaining them. Recent studies report that software projects are still running over time and budget to produce poor-quality software that do not meet requirements and are hard to maintain [NFG+06]. The main challenge is still to provide the appropriate theories, languages, abstractions, models, methods, and tools to assist software developers in building software.
→ In this context, the **approach** we have been following for several years consists in providing techniques and tools to advance the software engineering field in a **pragmatic** way, i.e. solutions that are intended to be easily grasped by average software developers and architects, solutions that follow and integrate well with the other trends of large and software intensive systems. Consequently our research work is mainly focused on providing **integration solutions that make some trade-offs**, mainly between reliability and flexibility. To do so, we constantly attempted to use and apply general principles of abstraction (design patterns, framework, models, models at runtime) and separation of concerns (compositional techniques).

In the field of software engineering, first paradigms and concepts such as structured programming, abstract data types and modularization were introduced with a clear objective of breaking the complexity, thus simplifying the engineering activity. Object orientation mixes this objective with a focus on controlled reuse through the open/close principle [Mey88]. In the mean time software applications grew in size to reach all departments of companies, and became interconnected inside a company. This leads to different approaches to again tame the resulting complexity. Approaches such as design patterns [GHJV94] and software frameworks [JF88] attempt to organize software abstractions so that different level of stability and reuse can be obtained from the software artifacts. First works on software architecture aimed at reasoning on the structure of a software system, its software elements and

their dependencies. Conversely an approach based on Aspects also emerged [KLM+97] with the aim of modularizing differently software systems. Refining the basic principle of *separation of concerns* to tame complexity [TOHS99], aspect orientation isolates supporting functionalities from the main business logic. Besides, to tackle the distributed nature of interconnected systems, a middleware layer was introduced and gained itself in complexity when interconnections were getting more and more complex. In parallel with these trends, some approaches focused on abstracting from technological peculiarities, such as Model-Driven Architectures (MDA) [KWB03], and Model-Driven Engineering (MDE) [Sch06], which generalizes the previous approach. The methodology is then to create and use domain models at all possible levels of the software development life cycle, fostering compatibility and simplifying design.
→ In our work, we use **model abstractions** in different forms, from abstractions into classes, some ad hoc models at run-time to software frameworks and integration in a model-driven toolchain.

At dawn of the new century, the concept of software component, already present in the first works on structured programming [McI68, Par72], finds a newly defined meaning with a parallel with the electronic industry. Basically a component defines both what it provides, and what it requires, making explicit some salient dependencies [BBB+00, Szy02]. Interestingly both the community of software systems, attempting to tame complexity of their middleware systems, and the software engineering community, striving to master the one of software, come to the same facts and similar proposals at the same time. Finally, the community was getting back to its roots, the foundation for the engineering of large systems. One aims at designing a system so that is composed of parts which, because they are smaller, should be easier to understand and build. One also defines interfaces that allow these parts to work together, be developed separately and maybe reuse in some way.
→ A first part of our work started with the shift to component-based programming [Szy02, CL02, LW07] and the objective to provide pragmatic solutions to design reliable and flexible component-based systems and frameworks with the use of **software contracts**, a concept well-defined in object-oriented systems [Mey92] (cf. section 1.2).

In the mean time a shift towards very large scale systems occurred. Software applications became naturally distributed inside and outside companies, not only through basic Internet protocols, but using the World Wide Web as a standard platform. Software intensive systems also begun to invade all aspects of businesses, societies and people. Hundreds of millions of people started to have pervasive access to a phenomenal amount of information, through different devices and from anywhere. Engineering these software systems means tackling the complexity of building and maintaining distributed and interconnected of 24/7 deployed applications. Mastering the evolution of software systems was an issue since the beginning of their development, but with computing tasks and applications executed over long periods of times, 24/7 in more and more cases, it clearly becomes crucial. Software systems must become more versatile, flexible and resilient by adapting to changing operational contexts, environments or system characteristics. To tackle this issue, a general approach is to provide adaptive capabilities to software, so that it can adapt at run-time to its changing environment (user requirements, faults, operational context, resource fluctuation) [LRS00, CLG+09]. With the aim of realizing the vision of autonomic computing [KC03], i.e. the application on a large scale of self-adaptivity to all software intensive systems, the field faces numerous challenges in engineering such self-adaptive systems [CLG+09, ST09].
→ A second part of our work then consisted in providing **self-adaptive capabilities** to our contracting system, through negotiation mechanisms over contracts and self-adaptive monitoring sub-systems (cf. section 1.3).

As for the evolution of software engineering trends, our work also concerns an approach that takes importance in the last decade. Facing the increasing demand of highly customized products and services, many complex variants must be produced and maintained, forming a new and important factor of complexity. Software Product Line (SPL) engineering can be seen as a paradigm shift towards modeling and developing software system families rather than individual systems [CN01]. Making the analogy of other industries such as automotive or semiconductor sectors, the approach aims at managing multiple similar software products by an explicit handling of common and variable parts.
→ Our work focused on Feature Models (FMs), a formalism first defined by [KCH+90] and now widely used in SPL engineering to model variability of all forms of artifacts and software sub-systems. Facing both the multiplicity and the increasing complexity of such FMs, our contribution consisted in applying the principles of *separation of concerns (SoC)* so to provide a set of sound and efficiently implemented **composition and decomposition operators for feature models**, as well as a Domain-Specific Language (DSL) for managing them on a large scale (cf. section 1.4).

1.2 Contracting Software

Historically, assertions were notably used to express program properties. An assertion in a program is a boolean expression that must be satisfied whenever the associated code is correctly executed. First works of Floyd [Flo67] and Hoare [Hoa69] concerned program construction and reasoning about their correctness. Several structured and modular programming languages also introduced assertions afterward. Abstract data types were then extended with preconditions and postconditions.
With object orientation, these assertions have been complemented with invariants on classes, making up specifications that were also called *contracts*. The Eiffel language [Mey92] was the first one to integrate these contracts and systematize their usage in the development life cycle, following a principle of "Design by Contract" [Mey92]. The specifications are then interpreted as mutual obligations and benefits, similar to business contracts, but between the developer and the user of a class. When contracts are checked at run-time, a failure can then be interpreted to precisely blame the responsible party. Clients (users of a class) are responsible for preconditions – they have to ensure that the precondition holds before calling the method –, while suppliers (developers of a class) are responsible for the postconditions and invariants – they should ensure postconditions and invariants hold whenever preconditions do –. Responsibilities make a clear metaphor to guide the design process and were also adaptable to inheritance, providing an interpretation as inheritance contracts between class designers. Moreover contracts can be well-organized with exception handling, separating correctness from robustness concerns, and can also be used as automated and up-to-date software documentation. Additionally contracts can be used in unit testing, so to check that it meets its contract assuming its subcontractors meet theirs. Embedding contracts and tests into components can finally make them *self-testable* [JDT01].
Different types of contracts are usually distinguished. A first classification was established by Beugnard et al. [BJPW99] when interpreting contracts to renewed forms of software components [Szy02]. Four levels of contracts were distinguished:

- ◇ *basic*, i.e. concerning syntactical properties (method names, parameter types) or simple semantic properties, e.g. interface definition languages;
- ◇ *behavioral*, which are related to properties expressed through pre/postconditions and invariants,

directly implementing the responsibility model of design by contract when method executions considered as atomic.

- ◇ *synchronization*, i.e. concerning interactions between components, typically defined by method call sequences following a specified pattern. Concurrency in method calls is then taken into account in this property description.
- ◇ *quality of service*, which finally encompass all contracts related to non-functional properties, e.g. response time, quality of information, etc.

In our work, we have first tackled the issues of contracting rich forms of software components, by developing a assertion-based contracting system for hierarchical components that goes beyond classic interface contracts. We afterward organized contracting systems as a framework abstracting both input formalisms and targeted platforms. Almost naturally, our work on software contracts led to the integration of contracting mechanisms inside an engineering process following a Model-Driven Architecture aimed both at service and component platforms. Additionally, we also explored the established relationship between our forms of contracts and testing, providing a framework to build self-testable hierarchical components.

1.3 Self-Adaptive Capabilities

Self-adaptive capabilities are provided by software systems to cope with changes at run-time. Self-adaptive software can be characterized by the fact that "it evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible" [LRS00]. This means that the self-adaptive capabilities should facilitate run-time decisions to control structure and behavior of the system. This latter is taking these decisions itself, with minimal or no human interactions, while reasoning about its own state and environment. The relevance of engineering self-adaptive capabilities in the software development landscape is due to the continuous evolution from software-intensive systems to ultra-large scale systems [NFG+06]. As acknowledged in [CLG+09], software systems must now become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics.
Properties of self-adaptive systems are now generally named self-* properties [SPTU05]. When IBM initiated their *Autonomic Computing* initiative [IBM01], they defined four main properties that serve as the *de facto* standard in the domain [ST09]: *self-configuring* is the capability of reconfiguring automatically and dynamically software entities in response to changes, *self-healing* consists in detecting, diagnosing, and reacting to disruptions and also in anticipating potential problems to prevent failures, *self-optimizing* is the capability of optimally and automatically managing performance and resource allocation, while *self-protecting* concerns the detection of security breaches and recovering from attacks. Actually *Autonomic Computing* [KC03] revisits the engineering of self-adaptive systems by aiming their application on a large scale to tame maintenance costs of all kinds of software intensive systems. One major challenge of the approach is the necessity to combine and evolve techniques and results from several research disciplines, e.g. artificial intelligence (planning, decision making, machine learning, agents,etc.), control theory, distributed computing and software engineering [ST09, CLG+09].
In order to organize self-adaptation, feedback control loops are recognized as one of the most generic mechanisms [CLG+09, BDG+09]. There can be several ways of presenting the key activities of a

feedback, but typically, it involves four steps (collect, analyze, decide, and act) [BDG+09]. In the IBM architectural blueprint for autonomic computing [KC03], the notion of autonomic manager is introduced. This is basically a component that implements a *MAPE-K* control loop. The name is an abbreviation for *Monitor*, *Analyze*, *Plan*, *Execute* and *Knowledge*. The loop is divided into four parts that share the knowledge and control a managed resource through sensors and effectors:

- ◇ The *monitor* function provides the mechanisms that collect, aggregate, filter and report details collected from the managed resource.
- ◇ The *analyze* function provides the mechanisms that correlate and model complex situations They allow the autonomic manager to learn about the system environment and help predict future situations.
- ◇ The *plan* function provides the mechanisms that build the actions needed to achieve goals and objectives. The planning mechanism is guided by policy information.
- ◇ The *execute* function provides the mechanisms that control the execution of a plan with considerations for dynamic updates.

In this context, a second part of our work concerned the adaptation of our contracting systems to some of these self-adaptation needs. We provided negotiation mechanisms, inspired from those conceived in multi-agent systems, which make it possible to adapt components or contracts at configuration and run times, with the aim to restore the validity of established contracts. We also designed and implemented a fine-grained support for a large class of non-functional properties within hierarchical software components, enabling their exploitation in the above negotiation process. Additionally, as contract checking and many self-management activity directly rely on appropriate monitoring, our work also comprised techniques and tools to provide adaptive monitoring systems.

1.4 Feature Modeling in Software Product Lines

A software product line (SPL) is "a set of software- intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [PBvdL05]. SPL engineering relies on the idea of *mass customization* [Pin99] known from many industries, like in the automotive, telecommunication, aerospace and avionics fields. Mass customization takes advantage of similarity principle and modular design to massively produce customized products. Taking its roots in the idea of program families [Par76], SPL engineering become popular in the 90s with the massive integration of software in families of electronic products such as mobile phones. Since then, many companies (Alcatel, Boeing, Hewlett Packard, Philips...) report significant benefits of using SPLs [Nor02].
SPL engineering is separated in two complementary phases. Domain engineering is concerned with development *for reuse*, and consists in analyzing the entire domain and its potential requirements, e.g. to scope the SPL and identify what differs between products, to identify reusable artifacts and plan their development, etc. On the other hand application engineering is the development *with reuse*, also called product development, in which concrete products are adapted to specific requirements and derived using the common and reusable artifacts developed in domain engineering. In the context of SPLs, MDE is gaining more attention as a provider of techniques and tools to tame their complexity of development. For example, generative software development techniques [CE00] aims at designing and implementing system families so that a given system can be automatically generated from a specification – a model – written in one or more textual or graphical domain-specific languages.

Central and unique to SPL engineering is the management of *variability*, i.e., the process of factoring out commonalities and systematizing variabilities of documentation, requirements, models, code, test artifacts... Variability is commonly described in terms of *features*, which are domain abstractions relevant to stakeholders (people concerned with the SPL). It is then usually modeled, using languages that can be graphical, textual or a mix of both.
Variability can be amalgamated into models [ZJ06, PVL+10] or be represented as first-class entities in meta-models, like in Clafer [BCW11] On the other hand, variability can be mapped to another metamodel [CA05]. This directly relates features and model elements and product models are derived by removing all the model elements associated with non-selected features. To realize variability at the code level, SPL methods classically advocate usage of inheritance, components, frameworks, aspects or generative techniques. At the model level, some approaches annotates a global model and a specific model is obtained by activating or removing model elements from a combination of features [CA05, BCFH10]. This is also referred as model pruning [SPHM09] or negative variability [VG07] Some other approaches, compositional, consists in separately implementing features in distinct software modules that are composed to obtain variants. Many techniques have been proposed to implement this form of positive variability [VG07]. In model-based SPL engineering, approaches composing multiple models or fragments have been proposed, relying on aspects [MVL+08], adapted superimposition techniques [AJTK09] or merging of class diagram fragments [PKGJ08b].
Considering approaches in which the variability description is expressed in a dedicated model, our work concerns *Feature Models* (FMs). First defined by [KCH+90], an FM is used to compactly define all features in an SPL and their valid combinations; it is basically an AND-OR graph with propositional constraints. This *de facto* standard is now widely used in SPL engineering to model variability of all forms of artifacts and software sub-systems. As FMs are getting increasingly complex, our work focused on applying the principles of *separation of concerns (SoC)* so to provide composition operators (insert, merge, aggregate) and a decomposition operator (slide) specific to FMs. These operators have a well-defined semantics that rests on the properties that must be preserved in terms of configuration set and hierarchy of the composed/decomposed FMs. Our work also consisted in creating a Domain-Specific Language (DSL), FAMILIAR, for managing FMs on a large scale. It enables one to combine the proposed operators with other reasoning and editing operators to realize complex tasks.

1.5 Outline

The remainder of this document is organized in three main chapters getting back to the research axes of our work.
Chapter 2 summarizes our activity on providing contracting techniques and tools in new forms of software architectures. We first describe ConFract, a contracting system using executable assertions on hierarchical components (section 2.1). Contracts are dynamically built from specifications at assembly times, then maintained at run-time and updated according to dynamic reconfigurations. Not being restricted to the scope of separated interfaces, new kinds of *composition contracts* are supported and semantically defined by their own responsibility model. Then the Interact framework is presented. It provides abstractions and automated mechanisms to facilitate software contracting with different kinds of specification formalisms and different component or service based architectures (section 2.2). This framework notably supports the integration of behavioral specification formalisms and relies on a central model handling both compatibility and conformance checking. The results of the ANR FAROS project are then described, showing how the central model of the Interact frame-

work was integrated and extended as a metamodel inside a model-driven toolchain ranging from high-level business constraints to contract checking mechanisms on different service and component oriented platforms (section 2.3). Finally, the chapter is ended by a presentation of an extension of our ConFract system to provide a complete contract-based built-in testing framework. This framework enables contracted components to be self-testable through appropriate embedded tests reusing contracts as oracles (section 2.4).
Chapter 3 presents the research conducted to provide self-adaptive capabilities in our contracting systems. We describe the extension of ConFract to support negotiable contracts in hierarchical components (section 3.1). The proposed negotiation mechanisms are inspired from similar mechanism in multi-agent systems, and allow for adapting components or contracts at configuration and run times. Reusing the responsibility model of contracts, components carry their own negotiation ability and can be dynamically reconfigured. A first concession-based policy is proposed in order to pilot the negotiation process for obtaining properties relaxation in contracts. Conversely, a effort-based policy is developed to direct the negotiation on the responsible component. The relations between negotiable contracts and autonomic control loops, as well as the use of the negotiation system to regulate itself are discussed. We then describe a model and a supporting run-time infrastructure that allows for reifying non-functional properties in relation with components, as well as for supporting a basic form of compositional reasoning that relate system properties to component properties (section 3.2). These patterns of non-functional properties can be exploited by the negotiation process presented before. The effort-based policy is then extended, enabling negotiation to be propagated according the compositional nature of some non-functional properties. Focusing next on the necessary monitoring features of current infrastructures, we propose a QoI-aware monitoring framework that is able to deal with multiple clients needing flexible and dynamically reconfigurable access to dynamic data streams with different Quality of Information (QoI) needs (section 3.3). The framework allows for instantiating monitoring systems with automatic configuration of all monitoring entities and data sources so that QoI and resource constraints are taken into account.
Chapter 4 presents our advances in the domain of feature modeling. Our proposed support for *Separation of Concerns* targeted to feature models is first presented (section 4.1). The support consists in a set of composition and decomposition operators with both a formal semantics definition and an efficient implementation. We notably define their semantics in terms of configuration set and hierarchy of the manipulated FMs. The *FAMILIAR* (*FeAture Model scrIpt Language for manIpulation and Automatic Reasoning*) DSL is then described (section 4.2). It enables one to combine the proposed operators with language constructs for importing/exporting FMs, editing FMs, reasoning about FMs (validity, comparison) and their configurations. The different constructs of the language are presented (variables, operations, scripts and modules). Several applicative case studies are also reported and discussed in terms of usage of the operators and the DSL (section 4.3). They range from consistent construction of scientific workflow to end-to-end handling of multiple variabilities in video-surveillance systems and reverse engineering of architectural variability.
Chapter 5 concludes this manuscript by assessing our results and discussing a research roadmap.

Main Supervisions and Publications

The results presented here are related to several PhD supervisions and publications. A complete list of publications is available at `http://www.i3s.unice.fr/~collet/publications.html`.
Works presented in chapter 2 have been published in several international conferences [CRCR05, DC06, COR06, COR07, CMOR07]. The ConFract system was realized under a first collaboration

contract with France Télécom R&D (now Orange labs) and partly through the Master Theses of Annabelle Mercier [Mer02] and Alain Ozanne. The resulting software was registered to APP and transferred to France Télécom R&D. The following framework, Interact, was developed in the context of Alain Ozanne's PhD Thesis [Oza07], which I co-supervised with Prof. Jacques Malenfant.
Works presented in chapter 3 have been the subject of several national and international journal and conference publications [CC05, CCOR06, CC06, CC07b, CC07a, LDCMR10]. The contract negotiation mechanisms were realized under another collaboration contract with Orange labs and concerns the Master and PhD Theses of Hervé Chang [Cha04, Cha07]. The results on the Adamo monitoring framework correspond to Bao Le Duc's PhD Thesis [LD10], which was co-supervised with Prof. Jacques Malenfant and funded by Orange labs.
Results of chapter 4 have been published in several international journals and conferences [ACLF09, ACLF10b, ACLF10a, ACC+11, ACLF11c, ACG+11]. They correspond to Mathieu Acher's Master and PhD Theses [Ach08, Ach11], co-supervised with Prof. Philippe Lahire.
The work evoked in the conclusion and related to engineering of feedback control loops are the subject of the ongoing PhD Thesis of Filip Krikava. Early results have been published in some international conferences [CKM+10, KC11].

CHAPTER 2

Contracting

Contents

2.1 A Contracting System for Hierarchical Components 10
2.1.1 The Fractal Component Model 10
2.1.2 Illustration 11
2.1.3 Rationale for Contracting Hierarchical Components 11
2.1.4 Specification with the CCL-J Language 13
2.1.5 The *ConFract* System 14
2.1.6 Implementation 18
2.1.7 Related Work 19
2.1.8 Summary 20
2.2 From a Contracting System to a Framework 21
2.2.1 Requirements for a General Contracting Framework 21
2.2.2 Case Study 22
2.2.3 *Interact* Framework Principles 23
2.2.4 Integration of Formalisms 25
2.2.5 Application to Executable Assertions 27
2.2.6 Application to Behavior Protocols 28
2.2.7 Kernel of the Contracting Model 32
2.2.8 Framework Roles 33
2.2.9 Contract Management in the *Fractal* Instantiation 34
2.2.10 Summary 35
2.3 From a Framework to a Model-Driven Toolchain 35
2.3.1 Motivations 36
2.3.2 FAROS Process Overview 36
2.3.3 Metamodels and Integration of Contracts 38
2.3.4 Illustration 40
2.3.5 Discussion 42
2.3.6 Summary 43
2.4 Contract-based Self-testable Components 44
2.4.1 Motivations 44
2.4.2 Illustration 45
2.4.3 Testing Framework Overview 47
2.4.4 Supported Testing Modes 48
2.4.5 Test Management and Framework Implementation 51
2.4.6 Summary 52

This chapter presents our research work on contracting techniques and tools, which have been conducted from 2002 to 2008.
This research starts in the context of the component definition being revisited [BBB+00, CL02, LW07] so to face the increasing complexity of more dynamic, evolving and long-living software systems. From McIlroy's appeal in 1968 [McI68], component-based software engineering (CBSE) has gone through an important evolution. Components were at first units of compilation, modules interacting through an explicit interface, then classes associated by use or inheritance links and finally, black boxes, organized in a (re-)configurable architecture and capable of communicating on networks through several interfaces.
One of the most used definition of this renewed form of component is given by Szyperski [Szy02]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.*" The notion of contract is explicitly part of the definition of software components, at least to define their interfaces. Our work then aimed at providing a contracting system for a rich form of software components that makes possible hierarchical compositions of them.

2.1 A Contracting System for Hierarchical Components

This section shares material with the CBSE'05 paper "A Contracting System for Hierarchical Components" [CRCR05] and the Euromicro-SEAA'05 paper "Fine-grained Contract Negotiation for Hierarchical Software Components" [CC05]. It mainly relates to work made in collaboration with Roger Rousseau.

2.1.1 The Fractal Component Model

Component-based programming aims at facilitating adaptable and manageable software development by enforcing a strict separation between interface and implementation and by making software architecture explicit [Szy02]. Coupled with meta-programming techniques, it can hide some non-functional aspects, like in mainstream component models (EJB, .Net, etc.) and their containers. At the beginning of the 2000s, both component-based frameworks and Architecture Description Languages (ADLs) provide means for explicit dependencies between components, but they only supported partially adaptation or extension capabilities [MT00]. There was thus a need to reconcile the advantages of the basic notions of software components, while having the means to manage the resulting architecture, to separate concerns (functional from non functional), to choose the right level of abstraction with components being created from other components, and to extend all these mechanisms.
The *Fractal* component framework [BCL+04, BCL+06] is a general and open component model that was designed to meet these requirements. It has the following main features: composite components (to have a uniform view of applications at various levels of abstraction), shared components (to model resources and resource sharing while maintaining component encapsulation), reflective capabilities (introspection capabilities to monitor a running system and re-configuration capabilities to deploy and dynamically configure a system) and openness (in the model, almost everything is optional and can be extended). The *Fractal* component model basically enables developers to hierarchically organize an application, with components being built from other subcomponents. Components can be connected through server (provided) and client (required) interfaces. The signatures of the interfaces are defined using the underlying language of the implementation `Julia` [BCL+04] of *Fractal*, currently `Java`.

Internally, a *Fractal* component is formed out of two parts: a membrane and a content. The content of a composite component is composed of other components, called subcomponents, which are under the control of the enclosing component. The *Fractal* model is thus recursive and allows components to be nested. The membrane embodies the control behavior associated with a particular component. In particular, it can *i)* intercept oncoming and outgoing operation invocations targeting or originating from the component's subcomponents, *ii)* superpose a control behavior to the behavior of the component's subcomponents or *iii)* provide an explicit and causally connected representation of the component's subcomponents. Different concerns of this control behavior are distinguished by controllers. Basic *Fractal* controllers are dedicated to manage life cycle (*LifecycleController*), component bindings (*BindingController*) and component content (*ContentController*).

2.1.2 Illustration

In the rest of this section, we use, as a working example, a basic multimedia player that has been developed with the Sun *Java Media Framework* API[2]. The architecture of the multimedia player is shown on figure 2.1 and presents a `FractalPlayer` component containing five subcomponents: `Player` which exclusively provides the playing service through its `start` method and manages some of its functioning parameters through attributes, `GuiLauncher` which manages the GUI part, `VideoConfigurator` which provides services to optimize the playing service (the `canPlay` method evaluates the ability to entirely play a video in its specific display size, according to available resources like the battery level), `Logger` which manages a history of played videos (the `lastUrl` method allows one to get the url of the most recently played video), and finally `BatteryProbe` that provides information on the battery (method `getLifePercent` returns the percentage of remaining stamina).
For all these components, their client interfaces manage what their environment should provide to realize their services. At assembly time, all these interfaces must be connected, through the content controller of the surrounding component (<fp> in our example), to interfaces of compatible type and of inverse role (client to server).

2.1.3 Rationale for Contracting Hierarchical Components

In component-based systems, like in object-oriented ones, it is well accepted that interface signatures, even with comments, are insufficient to capture and control the salient properties of an application [BBB+00]. More complete specifications are needed on the functional and extra-functional aspects (architecture, quality of services, etc.). Some properties can be checked early, using static analysis or proofs. Other properties, often extra-functional, which refer to runtime values, need to be dynamically checked. In the case of hierarchical components where the assemblies are dynamic, we liken the word *static* to *"before the component is (re-)started"*.
Either static or dynamic, many different properties can be expressed, using different specification formalisms [LBR99, dAH01, PV02, BS03]. For example, interface automata [dAH01] enables a specification to capture input assumptions about the order in which the methods of a component are called and output guarantees about the order of called external methods. Checking compatibility and refinement between interface models is then possible. Behavior protocols [PV02] express traces on interface method calls with a form of regular expressions and takes into account hierarchical components. These protocols can be defined on interfaces, frames (aka component types) and architectures (aka component internal assembly). Refinement of specifications are verified at design time, while

[2] http://www.oracle.com/technetwork/java/javase/index-142695.html

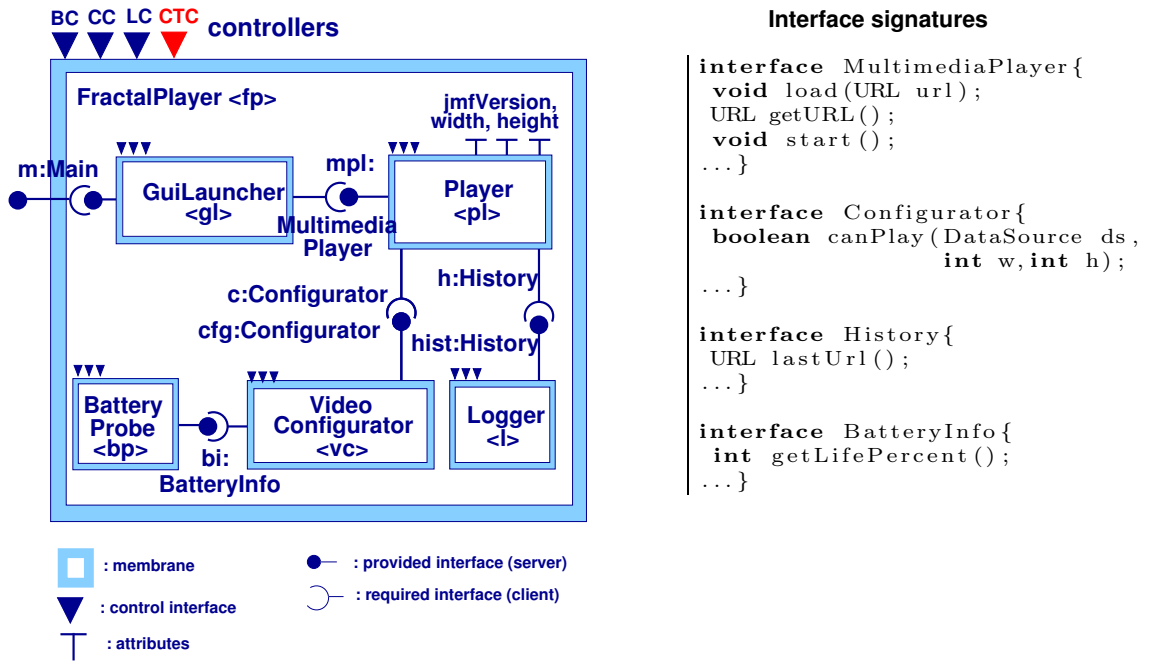


Figure 2.1: A multimedia player in Fractal.

adherence of a component's implementation to its specification is checked at run time. To build the *ConFract* system, we decided to first focus on the contracting mechanisms, rather than on the expressiveness of the specification formalism. The idea is to make explicit the contract at configuration[3] and execution times, in the same way as the architecture is made explicit in the *Fractal* model. The contract should then be a model at runtime that reifies the common definition of *"document negotiated between several parties, the responsibilities of which are clearly established for each provision"*.
When the *ConFract* development was initiated, to our knowledge, the proposals to explicitly support contracts for components all focused on interfaces or connectors, taken separetely. They aimed at specifying behavior [BS03], architectural constraints [Pah01] or quality of services [WBGP01]. As such they lack several important features to be well suited to our definition of components:

- ◇ Take into account a hierarchical assembly of components,
- ◇ Build contracts incrementally and update them if any dynamic reconfiguration occurs,
- ◇ Check them at configuration times or at least at runtime,
- ◇ Empower contracts with exploitable responsibilities, e.g. in case of violation.

Finally, we also advocate the contracts, as runtime objects, should be distinguished from the specifications, as input formalisms, they are built from. The overall objective of the *ConFract* is then to meet these requirements.

[3] As the *Fractal* model is open, we liken the configuration time to be a period that can encompass assembly and deployment, before a component is run, as well as a period of dynamic reconfigurations with re-assembly and deployment again.

2.1.4 Specification with the CCL-J Language

As one of main principles of *ConFract* is to clearly separate contracts from specifications, we developed an input specification formalism dedicated to our targeted component model. Executable assertions, the formalism first introduced with contracts [Mey92], are then used as they constitute a interesting trade-off between expressiveness and ease of learning and use.
The `CCL-J` language (*Component Constraint Language for Java*) is inspired by `OCL` [OMG97] and enhanced to be adapted to the *Fractal* model and its implementation in `Java`. Classic categories of specifications like preconditions (**pre**), postconditions (**post**) and invariants (**inv**) are supported. Some specific constructs like **rely** and **guarantee**[4] are also included but not discussed here. Each category consists of one or more clauses, identified by a number or a label and bound by a logical conjunction.
The main contribution of `CCL-J` is the provision of different scopes of specification that are adapted to salient location in an assembly of hierarchical components. Syntactically, the scope of specifications is adapted using variants of the **context** construct. It can refer to:

- ◇ a method of a `Java` interface: **context** *method-signature*;
- ◇ a component type: **on** `<Component_Type>` **context**...;
- ◇ or a particular component (instance or template of *Fractal* components [BCL+04]): **on** `<cpt_instance>` **context**...

As in current proposals for contracting components [WBGP01, BS03], it must be possible to use the connection point between two interfaces, client and server, to define some specifications. For example, the following precondition states that the input url should be valid for the `start` method of interface `MultimediaPlayer`, wherever it is used:

```
context void MultimediaPlayer.start()
  pre UrlValidator.isValid(getURL())
```

To express more relevant properties, it is necessary **to compose** external or internal properties by widening the scope, while respecting encapsulation, which is controlled by component membranes. The following specification shows an example of component type specification, with a specification of one of its interfaces in relations with the others. This specification defines both a precondition and a postcondition for the `start` method of the *Fractal* interface named `mpl` (of type `MultimediaPlayer`). The precondition also refers to another external interface of `<Player>`, the required interface named `c` of type `Configurator`, to express acceptable conditions to play the video. As for the postcondition, it refers to the required interface named `h` of type `History` and specifies that the last entry of the history matches the played video.

```
on <Player>
  context void mpl.start()
    pre c.canPlay(getUrl().getDatasource(),
                  <this>.attributes.getWidth(),
                  <this>.attributes.getHeight())
    post h.lastUrl().equals(getUrl())
```

[4] **rely**, resp. **guarantee**, states conditions that a method can rely, resp. must guarantee, during its entire execution.

All properties stated this way are located on *component types*, as they are valid whatever is their internal assembly. It must be noted that both specifications expressed until now refer to the `Multimedia-Player` interface, but the first one is general enough to be used on each binding of this type whereas the other one is just general enough to be interpreted on each instantiation of the `<Player>` component type.
Finally, in the case of a composite component, it is also necessary to define properties over its internals, accessible through its internal interfaces or through the external interfaces of its subcomponents. The specification below is a configuration invariant that constrains `<fp>` so that its subcomponent `<pl>` uses a version of the JMF API more recent than `2.1`. It uses specific constructs of *CCL-J*, such as parameters over specification and access to the attributes of a component (`getJmfVersion()`):

```
param jmfMin = JMF.V2_1
on <fp>
   inv <pl>.attributes.getJmfVersion().compareTo(jmfMin) >= 0
```

Another example of *CCL-J* capabilities is the usage of regular expression to denote several names referring to methods, interfaces and components. The following specification defines a 10% threshold for the battery, which is mandatory for the multimedia playing and which should be checked before any method is called in it (pattern $\star$):

```
on <fp>
  context <*>.*(*)
    pre  <bp>.bi.getLifePercent() >= 10
```

All the properties then concern component instances, as they are dependent from a specific assembly in the content of a composite component.

2.1.5 The *ConFract* System

Types of contract

The *ConFract* system distinguishes several types of contracts according to the specifications given by the designers.

- ◇ *Interface contracts* are established on the connection point between each pair of client and server interfaces and the retained specifications only refer to methods and entities in the interface scope. Our example of precondition on the `start` method of `MultimediaPlayer` interface is then used to build the interface contract of Figure 2.2 This contract is built on the binding between required interface `m:` `MultimediaPlayer` and provided interface `mpl:` `MultimediaPlayer`. The figure shows a textual representation of the corresponding contract object with all actual instances of interfaces and interfaces identified in the contract, as well as their responsibilities (see below).

- ◇ *external composition contracts* are located on the external side of each component membrane. They consist of specifications which refer only to external interfaces of the component. They thus express the usage and external behavior rules of the component. As shown on figure 2.3, the specification expressed on the `<FractalPlayer` component type is automatically taken into account to build the external composition contract on the `<pl>` component instance.

- ◇ *internal composition contracts* are located on the internal side of a composite component membrane. In the same way, they consist of specifications which refer only to internal interfaces

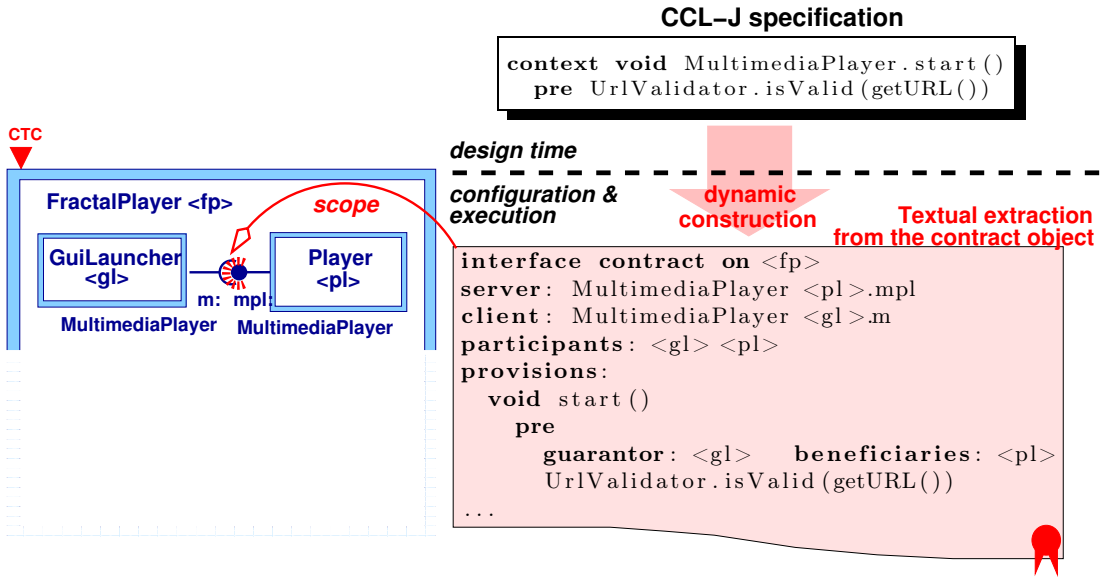


Figure 2.2: Example of interface contract.

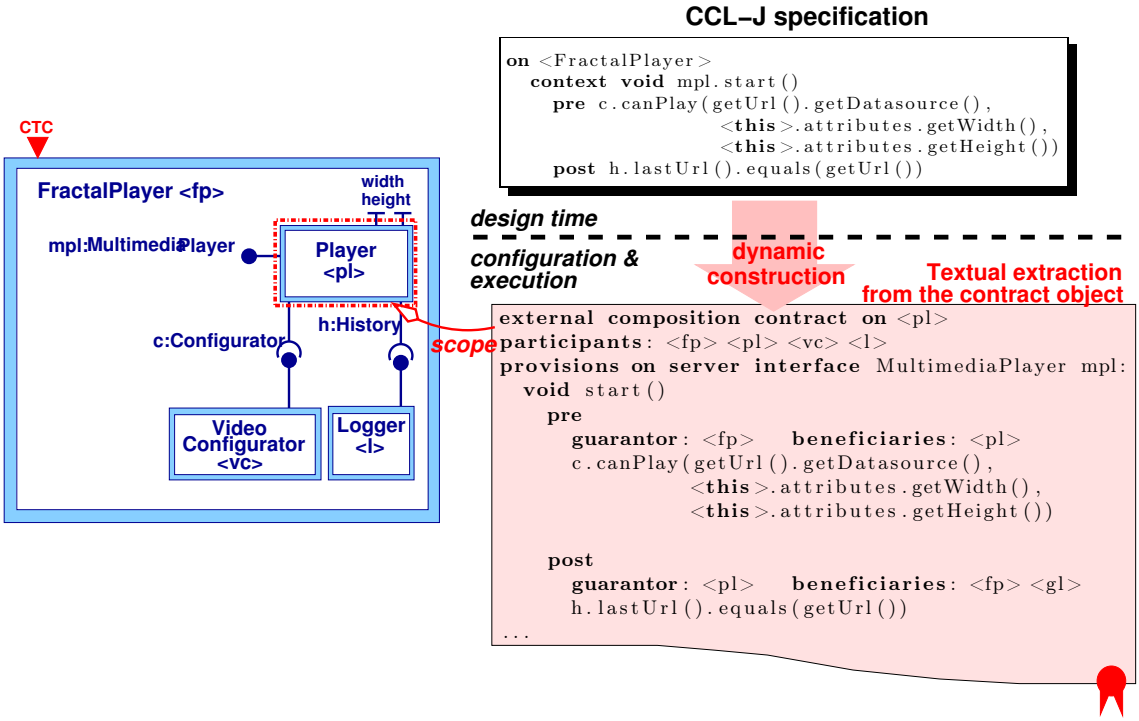


Figure 2.3: External composition contract on component <pl>.

of the component and to external interfaces of its subcomponents. This internal composition contract then enforces the assembly and internal behavior rules of the implementation of the composite component. In our example, this is the two specifications that we define on component <fp> (see Figure 2.4). One can also see that all specifications, even expressed separately, are grouped and interpreted in several contract provisions, a.k.a. clauses.

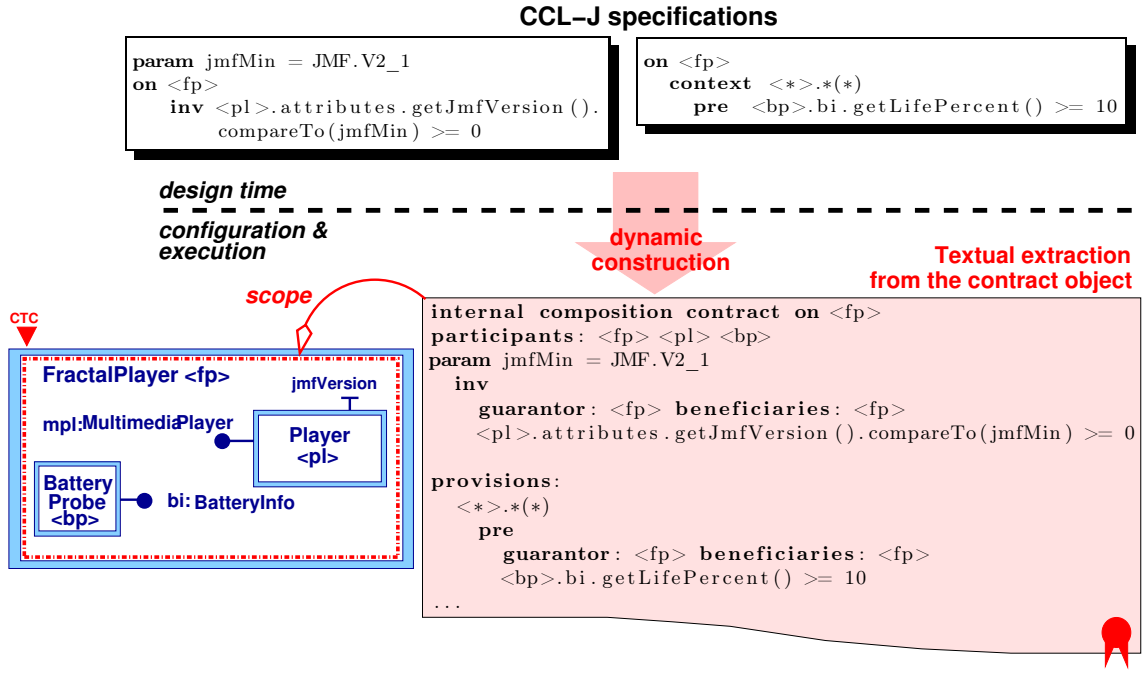


Figure 2.4: Internal composition contract on component <fp>.

Responsibilities

During the reification of a contract, the *ConFract* system determines the responsibilities associated to each specification, among the list of participating components in the contract. These responsibilities can be either *i) guarantor*, the component that must be notified in case of violation of the provision, and which has the capacity to react to the problem, or *ii) beneficiaries*, the components which can rely on the provision, or *iii)* possible *contributors*, which are components needed to check the provision, i.e. a contract is not complete without all its contributors identified. Contrary to most object-oriented contracting systems, there is no concept of blame or guilty party in our model, as it is more dynamic and open to negotiations. As a result, on a contract violation, the focus is more on how to dynamically adapt the application at best, preserving robustness, rather that on assigning blame about a correctness issue (see section 3.1).

In the case of an interface contract, these responsibilities are directly those of a client/supplier relationship as in an object contract [Mey92], as shown on Figure 2.5. The responsibilities for an external composition contracts are given on Figure 2.6. In this case, the interface role (client or server) directly impacts the interpretation of the responsibility. For server interface, the guarantor is the carrying component and beneficiaries are the surrounding component and components connected to this server interface. For example, on the component <pl> – the player – for the postcondition of a

construct	guarantor	beneficiary
pre	client <gl>	supplier <pl>
post	supplier <pl>	client <gl>

Figure 2.5: Responsibilities for an interface contract, with example of fig. 2.2.

method on its server interface mpl, the guarantor is the component itself, as it implements the method and provides the interface, and the beneficiaries are <fp>, which contains <pl>, and <gl> which is connected to the interface mpl. Conversely, the precondition on the server interface is guaranteed by the surrounding component, which is the only one able to ensure a property that is potentially related to several interfaces of one of its subcomponents. The component connected to the server interface only *see* this very interface and is not able to understand the complete composition contract. In our example of component <pl>, the precondition on mpl is guaranteed by <fp>, and not by <gl>, which cannot be responsible for a property dealing with the video configurator <vc> through the interface c.

interface role	construct	guarantor	beneficiaries
server *mpl*	pre	surrounding component <fp>	carrying component <pl>
server *mpl*	post	carrying component <pl>	surrounding + connected components <fp>, <gl>
client *h, c*	pre	carrying component <pl>	surrounding + connected components <fp>, <vc> (c), <l> (h)
client *h, c*	post	surrounding component <fp>	carrying component <pl>

Figure 2.6: Responsibilities for an external composition contract, with example of fig. 2.3.

As for the responsibilities associated to an internal composition contract, they are quite straightforward, as the composite component carrying the contract is at the same time the guarantor and the beneficiary in all cases. As this kind of contract is similar to some constraints put on its internal assembly, it is normal that the component is entirely responsible for its own implementation.

Progressive closure of contracts

When a component is inserted into an assembly, *ConFract* creates its internal composition contract if it is composite, and its external composition contract if it has some specifications bound to several of its interfaces. For every specification bound to some composition contracts, a provision *template* is created and attached to the composition contract. Every template is waiting for all its contributors to *close up*. When a new subcomponent is added into a composite, all the templates that participate in the concerned composition contract have their responsibilities completed. When all the contributors of a template are known, it is closed and becomes a provision. When all the provision templates of an internal composition contract are closed, the contract is *closed* as well, as all the responsibilities are

identified, and the component can be finally started.
For an interface contract, the life cycle is very simple, as there are only two participants in the contract. It is thus created during the connection between the interfaces and is automatically closed.
It must also be noted that the contract can simply be reopened when a dynamic reconfiguration occurs. If any binding of a component is removed[5], the corresponding contracts are reopened, e.g. both the interface contract between the bound interfaces and the external composition contract reopen. Similarly, if a component is removed from a composite, the internal composition contract of this composite also reopens, waiting for a new component to be added. This ensures that the contracts are always up-to-date and dynamically reflect any reconfiguration on the component architecture.

Contract checking

When building the contract, the *ConFract* system includes in each provision of a contract, the specification predicate (currently a `CCL-J` assertion), an interception context (the times and locations where the provision is supposed to be satisfied) and the necessary references to the context (component, interfaces, etc.). The contracts are then evaluated when the appropriate event occurs (see section 2.1.6).
At configuration time, the provisions of composition contracts that define invariant properties on components are checked, such as the invariant part of the internal composition contract of Figure 2.4. As for preconditions, postconditions and method invariants of all contracts, they are checked at runtime. When a method is called on a *Fractal* interface, the provisions of the different contracts that refer to this method are checked in the following way. Preconditions from the interface contract are first checked. As they are created from the client and server specifications, they also check hierarchy errors to ensure behavioral subtyping [FF01]. Preconditions from the external composition contract of the component receiving the call, are then checked, ensuring the environment of the component is as expected. Preconditions from the internal composition contract are then checked. It should be noted that preconditions from the three different kinds of contract are simply checked sequentially. No specific rule is needed to ensure substituability as the interface contract already defined it, and that the other preconditions are not sharing the same scope and responsibilities. A similar checking is done with postconditions and method invariants after the call.

2.1.6 Implementation

The *ConFract* system is integrated into *Fractal* using its reference implementation in `Java`, named `Julia` [BCL+04]. Julia is a software framework dedicated to components membrane programming. It is a small run-time library together with bytecode generators that relies on an AOP-like mechanism based on *mixins* and *interceptors*. A component membrane in Julia is basically a set of *controllers* and *interceptors* objects. A *mixin* mechanism based on lexicographical conventions is used to compose controller classes. Julia comes with a library of mixins and interceptors classes the programmer can compose and extend.

The contract controller

The various contracts are managed by *contract controllers* (`CTC` on Figures 2.2 to 2.4), located on the membrane of every component. As subcomponents are under the control of the enclosing component,

[5] In *Fractal*, a component must be stopped before any binding or content management.

every contract controller of a composite component manages the life cycle and the evaluation of the contracts that refer to its subcomponents and their bindings:

- ◇ the internal composition contract of the composite on which it is placed,
- ◇ the external composition contract of each of the subcomponents,
- ◇ the interface contract of every connection in its content.

During the creation of a composite component, the initialization of its contract controller creates its internal composition contract. The other contracts are built and updated by mixins.
According to the configuration actions made on components, the contract controller reacts as different *mixins* are placed on the other *Fractal* controllers:

- ◇ *Binding Controller* (BC). As this controller manages the creation and destruction of the connections between component interfaces, a mixin notifies the surrounding contract controller of connections (resp. disconnections) to instantiate (resp. to remove) the corresponding interface contract.
- ◇ *Content Controller* (CC). This controller manages the insertion of subcomponents inside a composite. A mixin notifies the contract controller of each insertion, so that it builds the external composition contract of the new subcomponent C. The contract controller also closes the provisions that refers to C in the internal composition contract. The inverse actions are realized during the removal of a subcomponent.
- ◇ *Life-cycle Controller* (LC). As the *Fractal* model is very open, the only moment when one can be sure that a component is completely configured is just before it is started, using the `start` method of the life-cycle controller. As a result, a mixin is added to perform "static" checks (*cf.* section 2.1.3). The contract controller of the component (resp. of the surrounding component) verifies that its internal composition contract (resp. external) is closed. Finally, the contract provisions that are statically verifiable, such as component invariants, are checked.

As for the evaluation of dynamic contract provisions, `Julia` interceptors are used. Every *Fractal* interface related to a contract receives an interceptor on its methods entry and/or exit. In the case of `CCL-J`, when a method is called on an interface, the contract controller is then notified and it applies the checking rules previously described.

2.1.7 Related Work

Since the *Eiffel* language, numerous works focused on executable assertions in object-oriented languages, notably for *Java* [LBR99, Plö02]. *JML* [LBR99] combines executable assertions with some features of abstract programs. It allows the developer to build executable models which use abstraction functions on the specified classes. `CCL-J` is much simpler than *JML* in terms of available constructs, but we only use `CCL-J` to validate the contracting mechanisms of *ConFract*. The composition contract provided by *ConFract* can be compared to collaboration contracts on objects proposed by Helm and Holland [HHG90]. The notion of views in the collaboration is similar to the roles of the participants in our contracts. However, in the *ConFract* system, the composition contracts are carried by components – which allows for distributing them in the hierarchy – and are automatically generated and updated according to the actions of assembly and connection.

Works on contracting components focused on using adapted formalisms to specify component interfaces. For example, contracts on *.NET* assemblies have been proposed [BS03], using *AsmL* as a specification language. Abstract programs are then interpreted in parallel with the code, but the contracts are only associated with interfaces. Numerous works rely on the formalism *QML* (*QoS Modeling Language*) [FK98b], for example to contract QoS related properties on components [LS04]. *QML* allows the designer to describe such contracts by specifying the expected levels of the qualities on interfaces, but does not allow one, unlike `CCL-J`, to combine functional and extra-functional aspects in the same specification (for example, it is not possible to link a extra-functional constraint to some input parameter of a method). Several works have also proposed contracts for *UML* components. In [Pah01], contracts between service providers and service users are formulated based on abstractions of action and operation behavior using the pre and postcondition technique. A refinement relation is provided among contracts but they only concerns peer to peer composition in this approach. In the same way, a graphical notation for contracting *UML* components is proposed in [WBGP01], focusing on expressing both functional (with *OCL* [OMG97]) and extra-functional (with *QML* [FK98b]) contracts on component ports. Here again, only the connection of components is considered and checking means are not discussed. More recently Defour et. al. [DJP04] proposed a variant of the contracts of [WBGP01] with *QML*, which can be used for constraints solving at design time.
ADLs have been proposed for modelling software architectures in terms of components and their overall interconnection structure. Many of these languages support formal notations to specify components and connectors behaviors. For example, Wright [AG97] and Darwin [Mag99] use CSP-based notations, Rapide [La95] uses partially ordered sets of events and supports simulation of reactive architectures. These formalisms allow to verify correctness of component assemblies, checking properties such as deadlock freedom. Some ADLs support implementation issues, typically by generating code to connect component implementation, however most of the work on applying formal verifications to component interactions has focused on design time. A notable exception is the SOFA component model and its behavior protocol formalism [PV02], based on regular-like expressions, that permit the designer to verify the adherence of a component's implementation to its specification at runtime. The extension of *ConFract* with such behavioral formalisms is the subject of a following work presented in the next section.

2.1.8 Summary

We have described the *ConFract* system, which proposes a contractual approach for hierarchical component models. Contract objects are dynamically built from specifications, at assembly time, and are updated according to dynamic reconfigurations. These contracts are not restricted to the scope of interfaces, taken separately. On the contrary, new kinds of contracts can be associated to the scope of a whole component. These *composition contracts* constrain either several external interfaces of a component, providing some kind of *"usage contract"*, or several interfaces inside the component, providing a sort of *"assembly and implementation contract"*.
In *ConFract*, the responsibilities are identified in a fine-grained way, at the level of each provision of a contract. As a result, developers can better organize violation handling and adaptations. The current implementation of *ConFract* follows the principle of separation of concerns by using *Fractal* controllers, which manage extra-functional services at the component level.
ConFract has been applied in different case studies, notably in a client/server application that organizes *instant* communities that share the same interest (see section 2.2.2). Different sub applications are then controlled inside a community, the Fractal player used as illustration begin one of them. This application will be further detailed in some of the following sections.

In the version presented, *ConFract* uses the executable assertions language `CCL-J` to express specifications at interface and component levels. This language allows the developer to express interesting properties at the component level, but other formalisms, especially oriented towards behavioral specification, have been identified as candidates for integration. A part of this integration in a more general contracting framework is the subject of the next section.
Besides, in order to better handle contract violations, the idea of some negotiation mechanisms have emerged from this work, mainly by exploiting the explicit responsibility model. This will be presented in chapter 3.

2.2 From a Contracting System to a Framework

This section shares material with the SC'06 paper "Enforcing Different Contracts in Hierarchical Component-Based Systems" [COR06], the SOFSEM'07 paper "Towards a Versatile Contract Model to Organize Behavioral Specifications" [COR07] and the SC'07 paper "Composite Contract Enforcement in Hierarchical Component Systems" [CMOR07]. It concerns Alain Ozanne's PhD Thesis and collaborative work with Jacques Malenfant and Nicolas Rivierre within a contract with France Télécom R&D (now Orange labs).
With the definition of components provided by models such as *Fractal* [BCL+04, BCL+06], contracts must not only be associated with connected interfaces between components, but also with their assemblies, so that they can organize the guarantee of properties related to exchanges between assembled components. With *ConFract*, described in the previous section, specifications with executable assertions are used to dynamically build contract objects at assembly time, which are maintained at run-time and updated according to dynamic reconfigurations. Not being restricted to the scope of separated interfaces, new kinds of *composition contracts* are supported and semantically defined by their own responsibility model.
Aiming at some reusable sets of abstractions to facilitate contract support in different contexts, we focused on determining a software framework [JF88] to do so. For our objectives, the two common definitions of a framework [Joh97] are relevant: "*a framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact*", and "*a framework is the skeleton of an application that can be customized by an application developer.*"

2.2.1 Requirements for a General Contracting Framework

Going beyond assertions, a general contracting framework should be able to interpret, in contractual terms, a larger class of formalisms. Indeed, either static or dynamic, many relevant properties can be expressed, using different specification formalisms [LBR99, dAH01, PV02, BS03]. Behavioral specifications are particularly complementary to execution assertions, which are state-based, because they allow for expressing constraints over method call order. For example, interface automata [dAH01] allows for checking compatibility and refinement between interface models that relate input to output call orders. Behavior protocols [PV02] express traces on interface method calls with a form of regular expressions and takes into account hierarchical components. These protocols can be defined on interfaces, frames (aka component types) and architectures (aka component internal assembly). Compatibility and refinement of specifications are verified at design time. Adherence of a component's implementation to its specification can be checked at run time or by program model checking [PPK06].

Combining different specification formalisms is then desirable to leverage reliability on component-based systems, but this task is rather complex, given the diversity of formalisms that express behavior, their numerous common points and differences, and the separation between static and dynamic approaches. This shows that a more general contracting framework should handle several formalisms, with the verification of conformance of a component to its specification, such as in our first contracting system, but also the compatibility between specifications.
Among contract models that have been proposed for components, all considered at least the conformance of the components' concrete realizations to their specifications. They handled quality of services [SAD+02, JDP03] or behavioral [TBD+04] properties, but were dedicated to one form of formalism. Fewer models dealt with the compatibility of components in a configuration. These models differed by their consideration of the architecture and by the point at which they explicitly define the compatibility. For example, in the CQML model [Aeg01], the compatibility property is not explicit. Even if an expression of the compatibility is provided in the language, it is not linked with the component architecture. On the contrary, the architectural configuration is taken in account in [TBD+04], but if the compatibility property is handled in the formalism, it is still not explicit in the model. Interface bindings are the bases of *Requirement/Assurance* contracts [Rau00], which explicit the compatibility of their participants, but the formulation of this property is dedicated to the contract formalism. The most advanced work in this direction was *Parameterized Contracts* [RSP03b]. Even if the compatibility is not explicitly part of this model, a generic expression is given with consideration to the architecture. Moreover behavioral and QoS properties, for specific formalisms, were considered.
Besides, the emergence of service oriented architectures [MLM+06], as well as the *Service Component Architecture* specifications [Ope07], put a strong focus on how to create service-based composite applications, leading to different architectural configurations to be considered. In this context, we also defined as a requirement of the framework that it must also enable software designers to use different architectures, with contract types tailored to the kinds of architectures it controls. We summarize these requirements in the four following properties for our framework:

P1 - *Make explicit the conformance of individual components to their specifications.*

P2 - *Make explicit the compatibility of components specifications (between components of same level of composition, and between a composite and its subcomponents), on the base of their architectural configuration.*

P3 - *Make explicit the responsibilities of participating components against each specification they are involved in.*

P4 - *Support various specification formalisms and verification techniques (at configuration or run times).*

The next paragraphs presents and illustrate the *Interact* framework, which meet these requirements.

2.2.2 Case Study

We illustrate our contribution on an instant messaging system with dynamic grouping capabilities. This application, named Amui[6], has been developed using both *Fractal* components and web service technologies. It served as a validating application for *ConFract* (see previous section), *Interact*,

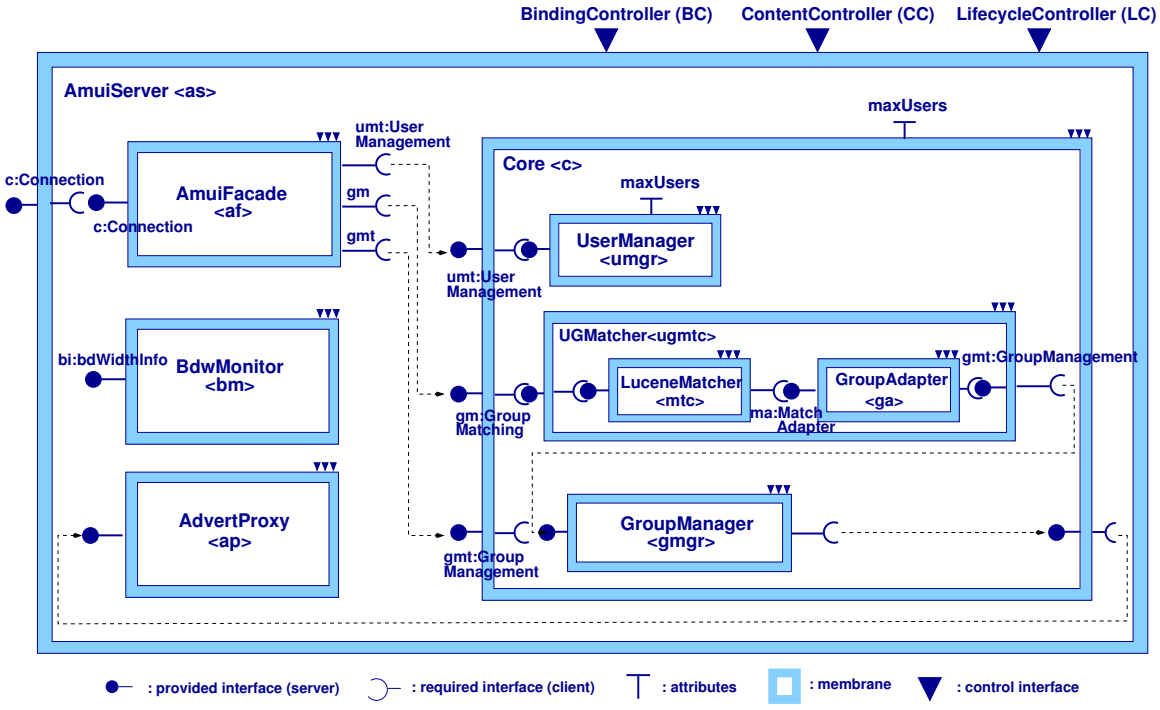
[6] *Amui* means *to gather* in Tahitian.

the framework now presented, and the contract negotiation system developed on top of them (see section 3.1).
The Amui system manages automatic grouping of users, according to their common interests, and dynamic application sharing. The server part is deployed on an Apache Tomcat container bundled with the Axis SOAP engine. Moreover, it also reuses instant messaging (IM) functionalities provided by the existing Openfire system[7], a cross-platform and extensible IM server that relies on the XMPP protocol. These functionalities are accessed and managed through component and service proxies, enabling the virtualization of the whole system through *Fractal* components.
Functionally, an user equipped with the Amui client application connects to the Amui server and gives some authentication information (login, password) as well as some keywords that describe its interests. The server then automatically finds the groups whose topics match the user's keywords, and it adds the user into the matched groups. Groups are associated with chat rooms and currently, once assigned in the same chat room, users can engage in a discussion with other users, and they also receive various advertisements according to their group topics. However, a larger range of shared functionalities can still be integrated into groups, for example to stream videos or simply to launch other applications on all clients.
Figure 2.7 shows a simplified view of the Amui architecture, focusing on the server part. The Amui server is structured with components at different levels of hierarchy. The top-level composite component named `AmuiServer` is formed out of three subcomponents : `AmuiFacade` which pilots all main functionalities, `Core` which encapsulates all the business functionalities, `Advert Proxy` which represents a proxy to the external advertisement web service used by the `Core` component and a `BdwMonitor` component to get information on the consumed bandwidth. The `Core` is piloted by `AmuiFacade` through three interfaces : one to match users' keywords to groups' topics named `GroupMatching`, one to manage users (`UserManagement`) and the last one to manage groups named (`GroupManagement`). Each interface is bound to one of the three components `UGMatcher`, `GroupManager` and `UserManager`, which simply provide functionalities corresponding to their name. In particular, `UGMatcher` contains the component `LuceneMatcher` that implements the matching mechanism by reusing the Apache Lucene[8] text search engine library. As it implements a rather general service, `LuceneMatcher` has a required interface, `MatchAdapter`, which gives access to a map-like structure in which the search is done. The map of information is then filled by the dedicated adapter component `GroupAdapter` with the various user groups information.
Many properties, both functional and non functional can be expressed on the extracted component assembly. In the remainder of the section, we focus on some specifications and contracts that illustrate our contributions.

2.2.3 *Interact* Framework Principles

In *Interact*, the overall design of the underlying contract model assumes that the collaboration between software entities (components, services, etc.) is driven by their architectural configuration. A complete and operational contract model is thus meant to verify properties of such configurations, and to determine the participating entities and their responsibilities. This implies that the used specifications should be explicit enough to allow a contracting system to determine the origin of a failure of a configuration. The model should also make it possible to express guarantees using various kinds of specification formalisms, provided that they can be interpreted in terms of contracts. On another

[7] http://www.igniterealtime.org/projects/openfire/index.jsp
[8] http://lucene.apache.org/



Interfaces Signature

```
interface Connection{

 int getMaxConnections();
...
}
```

```
interface CoreAttributes
          extends AttributeController{
 int getMaxUsers();
...}
```

```
interface BdWidthInfo{

 int getBdWLevel();
...
}
```

```
interface UserManagement{

 User createUser(String username, String password);
 void addUserTab(List users);
 int registrationQueueSize();
 void startUserCreation();
 void closeUserCreation();
 void setKeyWordList(Collection<String> keywords);
 void setGroup(User u, Group g);
...
}
```

```
interface GroupManagement{

 Collection<String> getGroupsId();
 Collection<String> getGroupTopics(String groupid);
 void startGroupSearch();
 void closeGroupSearch();
 Collection<String> getGroupTabForKeyword(String keyword);
 Collection<String> refineGroupSearch(String keyword);
 Group createGroup(String groupid);
 int removeGroup(Group g);
...
}
```

```
interface GroupMatching{

 Collection<String> searchGroupForKeywords(Collection<String>
     keywords);
 Collection<String> getGroupTab(Collection<String> keywords);
...
}
```

Figure 2.7: Architecture of the server.

hand, as different component and service models are to be handled, the contract model should be adaptable to different architectural styles, i.e. different configuration of collaborating components.

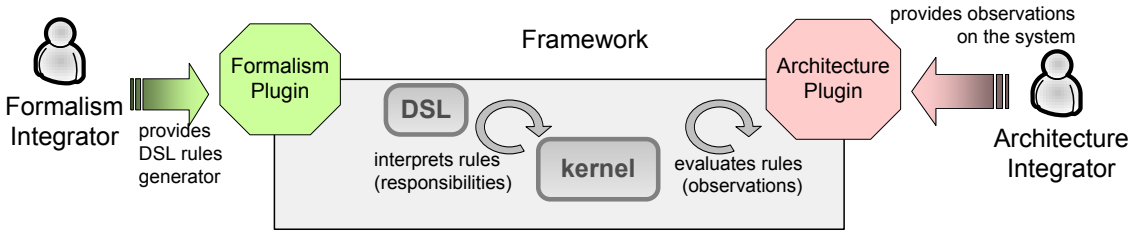


Figure 2.8: Principles of the *Interact* framework.

As shown on figure 2.8, the framework is based on a central contracting model, that acts as a reasoning kernel. A *contract* is thus defined as a software artifact that takes as input an architectural configuration and some specifications related to the components of its configuration, and evaluates the validity of the configuration, with a precise diagnosis in case of failure. To handle the variability of formalism, the integration of each formalism is made by providing a rule generator for the language. Rules are expressed through a provided DSL, described in the next paragraph. As for the architectural part, a specific plugin is also provided so that the DSL rules can be interpreted according to events occurring on a targeted architecture.
In the kernel, the contract model integrates specifications following an assume-guarantee semantics [AL93]. As this semantics basically describes commitments between a component and its environment, it enables the contracting system to make explicit the assumptions and guarantees and to compute both conformity and compatibility checking on the manipulated specifications. Figure 2.9 shows the concrete syntax associated with the generic contract model. It mainly defines:

Participants. The participants of a contract are the components of the architectural configuration it constraints.

Clauses. Each of the clauses is associated to a contract participant and contains a guarantee and an assumption following the assume-guarantee semantics. The associated participant is responsible for the satisfaction of the guarantee as long as the other participants, its environment, satisfy the assumption. More precisely the guarantee constrains elements that are provided by the responsible participant (its emitted signals...), whereas the assumption constrains elements that are required by the participant (incoming signals...).

Agreement. The agreement makes concrete the compatibility between the different clauses of participating components. Components work together by exchanging elements they provide and require. Their collaboration then requires that a guarantee made by the provider of an element on this latter fulfills the assumption made on this element by its requiring component. The agreement is thus the combination of these compatibility expressions on the exchanged elements.

The underlying kernel object model is discussed in section 2.2.7.

2.2.4 Integration of Formalisms

The integration of a formalism in the contracting framework is concerned by the definition of appropriate verification methods and tools. The provided DSL enables integrators to focus on the semantics

```
Contract
{
  Participants : <component_name>* ;
  Clauses :
  {
   clause : <name>
     responsible : <component_name>;
     guarantee : <expression>;
     assumption : <expression>;
   }*
  Agreement : { agreement expression }
}
```

Figure 2.9: Concrete syntax pattern of a resulting contract object.

of observations and verifications rather than on technical particularities. Using the DSL one can give a set of rules describing where and when observations occur, what values they capture, and the verifications to be triggered. A rule is defined the following syntax pattern:

```
On <a component>
  Observe {
       ( val: <some value> at: <some times>; )+
  }
  Verify <some properties>
```

The *On* block defines what spatial domain of the system is visible to the rule, i.e. a component scope. The *Observe* block describes the observations operated in the scope. It contains a list of observations that are defined by the statements *val:... at:...*, where the *at* block gives the times at which the value described in the *val* block can be observed. Finally, the *Verify* block describes the checking part, where the property to evaluate is a predicate that takes the *val* values as parameters.
In order to define when observations should occur, some atomic observable events are provided. For example, the following definition contains one event: the entry in the *getGroupsId()* method of the *gmt* interface (see the Amui example on Figure 2.7).

```
  at: entry Collection<String> gmt.getGroupsId()
```

Basic regular expressions enable designers to easily denote sets of events that encompass several method calls or several interfaces using wilcards.

```
  at: entry * gmt.*(*) , exit * gmt.*(*)
```

This set of events contains all events that are determined by an entry or exit of any method on the *gmt* interface. This kind of event specification is similar to what is used in aspect-oriented programming [KLM+97]. In our context, events that are specific to components life cycle are also manipulable so that configuration events can be taken into account. For example, adding/removing a component to/from a composite one, or binding/unbinding two interfaces. As the *Fractal* platform provides these control features through extensible interfaces [BCL+04], it is quite straightforward to be notified of these events in this architecture. In the general case, this is the responsibility of the architecture pattern to trigger relevant events to the contracting kernel so that they can be used in rule interpretation.

2.2.5 Application to Executable Assertions

We now show how the *CCL-J* assertion language (cf. section 2.1.4) we developed for the *ConFract* system (cf. section 2.1) can be integrated in the framework by providing a appropriate DSL generator. We give examples of some *CCL-J* specifications on the Amui system and show the generated contracts.
Regarding interface specifications, the following assertion concerns the *createUser* method on the *UserManagement* Java type:

```
context User UserManagement.createUser(String username,
                                       String password)
 pre: !username.trim.equals("")
```

In this particular case, the system can produce the equivalent of an *interface contract* in *ConFract* wherever this interface is used on the binding between a client and a server interface. In the Amui server, this is for example the case between the two interfaces connected between the AmuiFacade <af> and the Core component <c>.
The following text shows a textual representation of the contract object, using the resulting DSL rules to define observations:

```
Contract :
  Participants : <af>, <c>;
  Clause :
    responsible : <af>
    guarantee : rule {
      On <c>
      Observe {
        val: username
        at:  entry umt.createUser(String username,
                                  String password);
             }
      Verify !username.trim.equals("")
  ...
```

Here, the responsibilities defined for *ConFract* are reproduced with *(i) guarantor* which is responsible to ensure the clause and must be notified in case of violation, *(ii) beneficiaries* which can rely on the clause or *(iii)* possible *contributors* which are needed to check the clause. The syntax above only shows the responsible component, in our example, this is a classic client-server relationship: the component <af> is thus responsible to ensure the precondition as it is the caller of the method.
To illustrate contract based on exchanges between components, we define the following specifications:

```
on <af>
 context void umt.addUserTab(List users)
  pre:
   users.size() < c.getMaxConnections();

on <c>
 context void umt.addUserTab(List users)
  pre:
   users.size() < 20-umt.registrationQueueSize();
```

The first precondition above is a specification on the <af> component, which states that the size of the list of registration requests does not exceed the number of simultaneous user connections accepted by the server. The second precondition, on component <c>, constrains the size of the registration list received so that the final size of the registration queue is less than 20[9]. The resulting contract, which will be managed by the contracting system, has the following textual form:

```
Contract :
 Participants : <af>, <c> ;
 Clause :
  responsible : <af>
   guarantee : rule {
    On <af>
      Observe : val : users at entry umt.addUserTab(List users);
      Verify : users.size()<c.getMaxConnections();
   }

 Clause :
  responsible : <c>
  assumption : rule {
   On <c>
     Observe : val : users at entry umt.addUserTab(List users);
     Verify : users.size()<20-umt.registrationQueueSize();
   }

 Agreement :
   On <c>
     Observe :
      val : users at entry umt.addUserTab(List users);
     Verify :
      users.size()<c.getMaxConnections() =>
         users.size()<20-umt.registrationQueueSize();
```

The contracting system is thus able to interpret these two preconditions in the terms of the underlying contract model. It follows that in the first clause, <af>, client of *addUserTab* method, classically guarantees the first precondition (property P1 and P3). It also comes that <c>, server of the *addUserTab-method*, assumes then the second precondition, that is the second clause of the contract (property P1). This assembly of two components is valid not only if their clauses are respected (property P1), but also if these latter are compatible. This compatibility property constrained what is exchanged, i.e. the users list, and is made explicit by the agreement. It represents the implication of the assumption by the guarantee, i.e. the assumption of component <c> on a received users list must be fulfilled by the guarantee of component <af> on this sent list (see Figure 2.10). The inverse implication would have been built if postconditions were also present in the input specifications.

2.2.6 Application to Behavior Protocols

The behavior protocol (BP) formalism [PV02] allows one to specify and verify the correctness of communication among components. It has been applied to hierarchical component models (SOFA [PV02] and Fractal itself[10] [KAB+06]) to capture both horizontal (client-service) and vertical (nest-

[9] This kind of constraint has been really observed on the underlying IM server, with too many simultaneous user registrations being canceled by the system.

[10] http://fractal.objectweb.org/fractalbpc/index.html

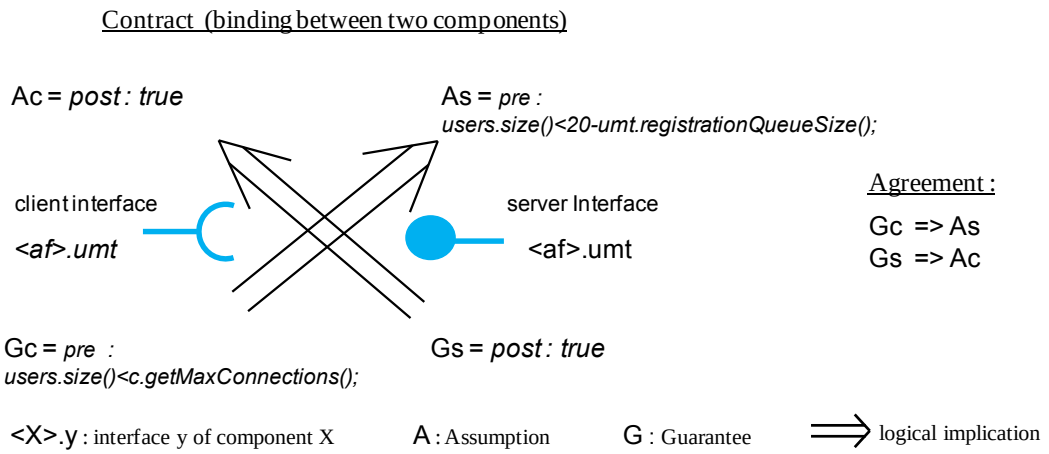


Figure 2.10: Agreement of the assertion-based contract between <af> and <c>.

ing) communications among components. We now illustrate how the framework would integrate and interpret BP in terms of explicit contracts.

BP are associated to the frame and architecture of components. A frame is a black-box view of a component, defining its provided and required interfaces. An architecture describes the structure of a composite component. It is formed by the frames and bindings of its sub-components at the first level of nesting. A BP is an expression describing a set of traces (sequences of events). When applied to components, every method call or a return from a method call forms an event. The notation proposes several useful shortcuts. For example, the notation *!i.m* describes the activity of a caller component (emitting a *m* method call on its required interface *i* followed by accepting the return), while *?i.m* describes what the callee component does (accepting the call on its provided interface *i* and emitting the return). Some of the operators employed in behavior protocols are *;* for sequencing, + for alternative choice, * for finite repetition, and | for parallel interleaving of the traces generated by the operands. If *P* is an arbitrary protocol, *?i.m{P}* means that the call request of *m* is absorbed, and while *m* is processed, the component behaves as specified by *P*; afterwards, the call response of *m* is emitted.

The frame protocols of the <ugmtc>, <gmgr>, <umgr> components and the composite <c> of the Amui system (cf. section 2.2.2) can be completely specified. We now only describe a part of it to illustrate our contribution. In order to associate the new users to their relevant groups, the *getGroupTab* method is called on the matcher component for each user. This component then retrieves group information through the group manager. Considering the specification *ugmtcFP*, the frame protocol of the matcher component <ugmtc>, means that while its method *getGroupTab* is processed, this component emits several calls to get and refine group searches. More precisely, <ugmtc> accepts a *getGroupTab* call on its provided interface *gm*. While this method is processed, this component emits a sequence of calls on its required interface *gmt* (*startGroupSearch*, a finite number of *getGroupTabForKeyword*, a finite number of *refineGroupSearch* and finally *closeGroupSearch*). Afterward, the response of *getGroupTab* is issued.

```
// ugmtcFP : frame protocol of the UGCMatcher component
?gm.getGroupTab{
  !gmt.startGroupSearch;
  !gmt.getGroupTabForKeyword*;
```

```
  !gmt.refineGroupSearch*;
  !gmt.closeGroupSearch
}
```

The following protocol concerns the *GroupManager* component, which behavior has three alternatives. The first one is symmetric to output of the protocol described above and the two others relates to group creation and removal.

```
// gmgrFP : frame protocol of the GroupManager component
(?gmt.startGroupSearch;
 ?gmt.getGroupTabForKeyword*;
 ?gmt.refineGroupSearch*;
 ?gmt.closeGroupSearch
)
+ ?gmt.createGroup
+ ?gmt.removeGroup
```

```
// umgrFP : frame protocol of the UserManager component
?umt.startUserCreation;
?umt.setKeyWordList;
?umt.setGroup*;
?umt.closeUserCreation
```

The following protocol finally expresses the composite protocol of the *Core* component, which is refined by the three previous protocols.

```
// cFP : frame protocol of the Core component
?umt.startUserCreation;
?umt.setKeyWordList;
?gm.getGroupTab;
?umt.setGroup*;
?umt.closeUserCreation
```

Using the behavior protocol formalism and its associated tools, the designer can verify:

- ◇ the adherence of a component's implementation to its specification. This kind of verification refers to our property P1 and can be achieved using an appropriate runtime checker or program model checker tools.
- ◇ the correctness of composing or refining the specifications. This kind of verification refers to property P2 and can be achieved using model checker tools: either horizontally (checking that frame protocols cooperate well together when composed at the same level of nesting) or vertically (checking the compliance between the architecture and the frame protocols of a composed component).

In our case, as an architecture protocol of a composed component is constructed as a parallel composition of the frame protocols of its sub-components, the architecture protocol of the Core component <c> is built as (*ugmtcFP* ⊓ *gmgrFP* ⊓ *umgrFP*), where ⊓ is the parallel composition.
Regarding the contractual interpretation of Behavior Protocols, we observe that a frame protocol specifies the valid traces of incoming and outgoing calls of a component. It naturally fits with the notion of clause and makes it possible to express the conformance of a component to its specification and its responsibility (as long as its incoming calls satisfy the protocol, its outgoing calls must satisfy

this protocol). Moreover the compatibility of behavior protocols can be checked between nested components, or between components at the same level of nesting. This makes it possible to express the compatibility based agreement of a contract on any configuration of components.
We now illustrate how our properties P1, P2 and P3 can be expressed in our contracting framework.
Conformance (P1) and Responsibility (P3). The illustration applies to the component <ugmtc> and its frame protocol *ugmtcFP* introduced above. The verification of the protocol is separated in two rules, one for the guarantee and one for the assumption. The guarantee rule verifies that each call of the component <ugmtc> is emitted in conformance with the protocol, while the assumption rule verifies that each received call is effectively expected by the protocol. The guarantee must hold as long as the assumption holds, otherwise the component <ugmtc> violates its specification. A runtime checker tool is used to verify the conformance at each new observation.

```
Contract :
  Participants : <ugmtc>, <gmgr>, <umgr>, <c> ;
  Clause :
    responsible : <ugmtc>
    guarantee : rule {
      On <ugmtc>
        Observe : val : at entry gm.*;
        Verify : runtimeCheck(ugmtcFP);
    }
    assumption : rule {
      On <ugmtc>
        Observe : val : at entry gmt.*;
        Verify : runtimeCheck(ugmtcFP);
    }
    ...
```

Compatibility (P2). The next illustration relies on a component life cycle event. The rule expresses that, just before starting the composed component <c>, an agreement must be verified between this component and its subcomponents [11]. The verification is done first horizontally (checking that the parallel composition of the frame protocols of the components <ugmtc>, <gmgr> and <umgr> is valid) and then vertically (checking the compliance between the architecture protocol obtained from the first step and the frame protocols of the composed component <c>). This kind of verification, based on model-checking, can be alternatively realized at the ADL level.

```
Contract :
  Participants : <ugmtc>, <gmgr>, <umgr>, <c> ;
  ...
  Agreement :
    On <c>
    Observe :
      val : at : entry <c>.start
    Verify :
      verticalCheck (cFP,
                     parallelCheck (ugmtcFP, gmgrFP, umgrFP))
```

[11] As in *ConFract*, static checking on *Fractal* component can only be done when one is sure an assembly is complete. We use the *start* method here, but another integration could have associated this verification to some external event, such as an explicit validation inside an IDE

It must be noted that the methods used in the `verify` clauses (*runtimeCheck, verticalCheck, parallelCheck*) are to be provided when integrating the BP formalism in the framework. They encapsulate appropriate checking methods to be accessible in the DSL rules.

2.2.7 Kernel of the Contracting Model

We now describe the object model of our contracting kernel. Following our main design choices, a contract must be represented a set of clauses constraining its participants, that are finally bound by an agreement. Figure 2.11 shows a class diagram of the main reified concepts. A *Participant* is an object

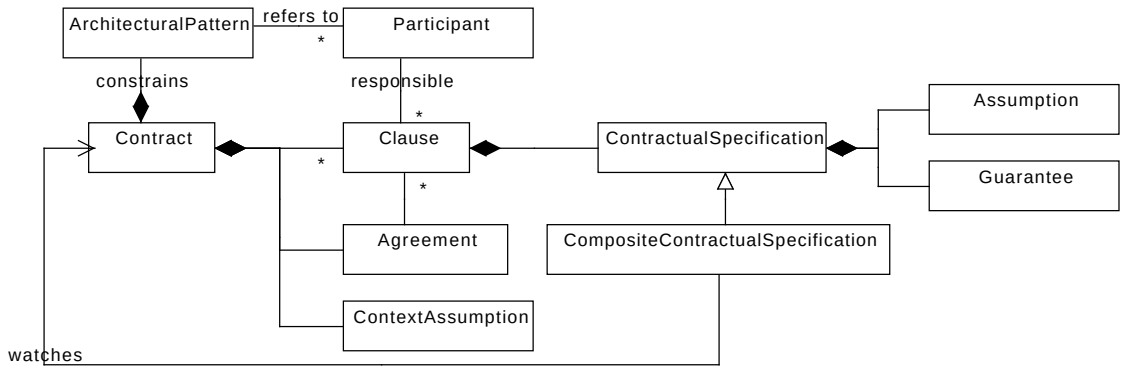


Figure 2.11: Kernel model of the contracting framework.

that refers to a compositional entity of the architecture, a *Fractal* component in the specialization of the framework for *Fractal* that we have used so far. Participants are obtained by the *ArchitecturalPattern* described below. *Guarantee* and *Assumption* hold a predicate and the description of the observations on the system it constrains. For a given component, the guarantee constrains what it provides (its emitted messages...) while the assumption constrains what it requires (its received messages...). According to the formalism used, their satisfaction can be evaluated at configuration or run times. A *ContractualSpecification* is simply a representation of the predicate that binds together an assumption and a guarantee for a given component. It follows the assume-guarantee principle: as long as the assumption is true then the guarantee has to be also true [AL93]. Consequently, our model applies to specification formalisms that are modular, i.e. a specification can be attached to a component, and that can also be interpreted in assume-guarantee terms (property **P4**).
A *Clause* is an object associating a contractual specification with a participant of the contract (property **P1**), which is then responsible for its guarantee (property **P3**). The model relies on architectural paths, not detailed here, to navigate in the component structure. They allow a *Clause* to enforce its specification, by checking if its guarantee and assumption denote respectively observable actions under the control of the component it constrains (e.g. emitted calls) or of its environment (e.g. received calls). Some strategy objects are associated to the evaluation of clauses, to detect if a guarantee is violated before its associated assumption. An *Agreement* then expresses the compatibility of the clauses (property **P2**) of the contract, stating that the assumptions of the collaborating parties are fulfilled by their guarantees in a given environment. An *ArchitecturalPattern* defines a configuration of relations between software entities. It is used to discern the entities and their relations constrained by

the contract. A *ContextAssumption* expresses a supplementary assumption on the environment of the components, i.e. a predicate constraining the context in which they are composed.
In addition the model also supports the *vertical composition* of contracts, still relying on the Abadi/Lamport theorem [AL93], which makes explicit, under some assumptions, the dependency between the specification of a component and the ones of its subcomponents. Enforcing vertical composition consists in checking the compatibility between the specification of a composite and the composition of its subcomponents specifications. In order to support this, A *CompositeContractualSpecification* class refers to a *Contract* between its subcomponents. This contract reifies an agreement so that the resulting composite contract can check the validity on the agreement between subcomponents. The *CompositeContractualSpecification* is itself an assume-guarantee *ContractualSpecification* obtained from the specifications of its subcomponents, which make explicit the conclusion of the theorem rule. The application of this inference rule relies on several prerequisites, which are going to be either ensured by construction, or checked on the components by appropriate elements in the contract model.

2.2.8 Framework Roles

At each stage of the framework usage is associated some roles that we now describe, with a focus on the instantiation of the framework on the *Fractal* platform (cf. figure 2.12):

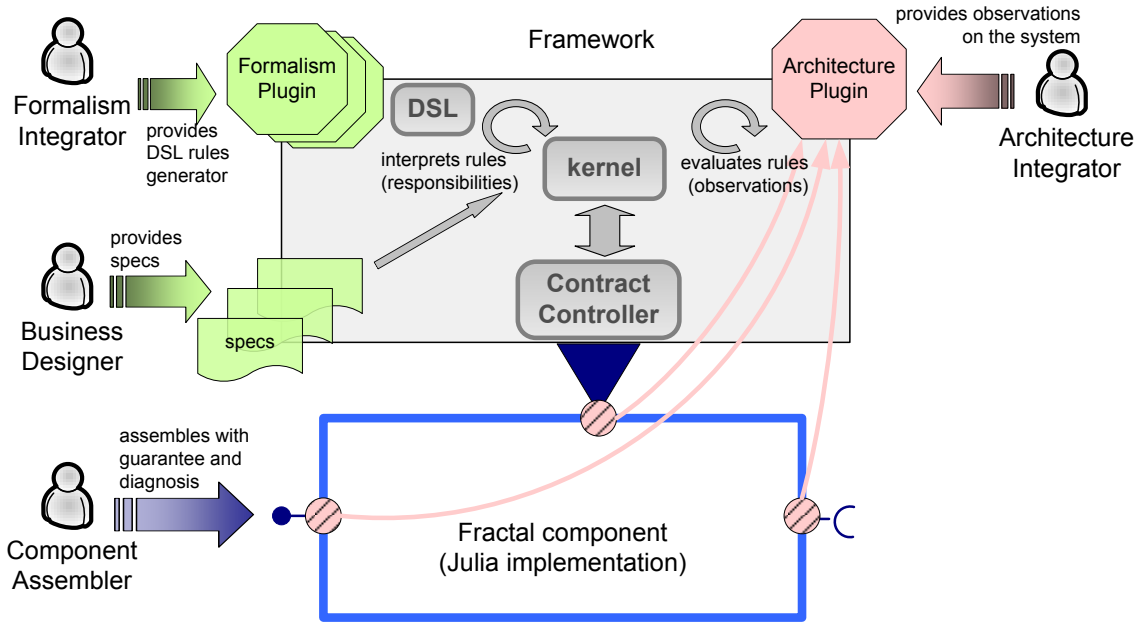


Figure 2.12: Roles and features of the framework with a Fractal instantiation.

On the formalism side, each *formalism integrator*, an expert in the semantics of the formalism to be integrated, has to provide a plugin that is able to parse specifications and to associate them to DSL rules, as described in section 2.2.4. On the architecture side, an *architecture integrator* provides observations on the system, so that the kernel can reason on an architecture. This encompasses introspection on architectural elements (components, interfaces in the case of *Fractal*) and behavioral observations (description and notification of observed events, such as method calls on *Fractal* interfaces).
Later on, *business designers* can use available formalisms to write specifications on components and their assemblies. At configuration and runtimes, *component assemblers* can simply put together

specified components. The contracting framework is then able to build contracts managed by contract controllers (see next section). Through these controllers, component assemblers get information on the constructed contracts, as the kernel can compute agreements and shows responsibilities among assembled components.

2.2.9 Contract Management in the *Fractal* Instantiation

To present the framework, we have only focused on its *Fractal* instantiation as it is the richer form of architecture that we have targeted and was the only one to be validated end-to-end. We briefly discuss here how contracts are managed at runtime in *Fractal*, mainly by showing the differences and extensions from the *ConFract* system implementation described in section 2.1.6.
To do so, the framework provides interfaces and abstract classes to provide *contract manager*. In `Julia`-based implementation of *Fractal* that we use, this is an adapted form of the contract controller presented in the previous section that will implement the contract manager interface. Figure 2.13 gives an overview of the main interactions realized by a contract controller. Basically, the con-

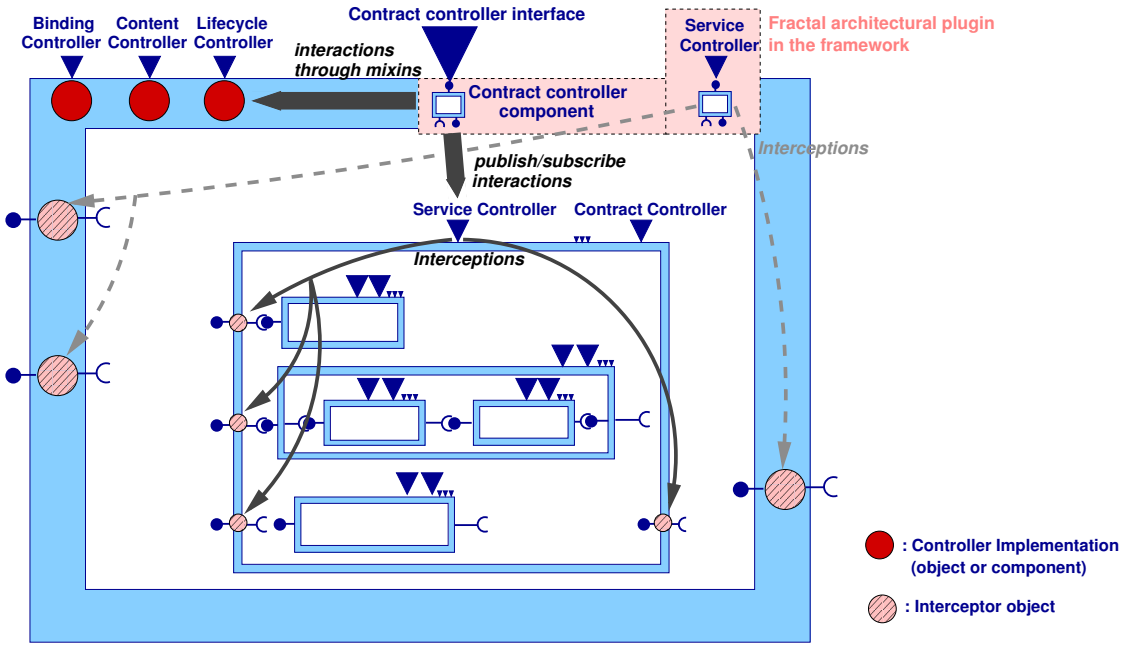


Figure 2.13: Main interactions related to the contract controller in the framework.

tract controller still implements the same functionality as in *ConFract*, but in the framework, each modification of the architecture perceived by one of the other controllers (binding, content, life-cycle) is now transformed in an event handled by the framework DSL engine. Contract objects, as shown in previous paragraphs, are incrementally built and attached to the corresponding contract controller. For the runtime checking part, which concerns functional interfaces (CCL-J assertions checking, observations on Behavior Protocols), the architecture plugin is realized through another controller, named *ServiceController*. It pilots interceptor objects put on each functional interface of its component. An object or component can then register on this controller to be notified of specific events (e.g. entry of a method). This mechanism directly implements the necessary observation part of the architectural plugin. Consequently, when a contract is built through the contract controller, it registers itself to the

service controller of each component that has to be observed. At runtime, when a message is sent to an interface, the service controller detects it through one of its interceptor objects, notifies the contract controller, which evaluate the appropriate rules, thus checking clauses and agreements.

2.2.10 Summary

We have described a general framework that aims at abstracting the integration of different specification formalisms so that given specifications can be organized as contracts on different software architectures with an appropriate support for checking. The nature of the proposed framework is expressed through two major plugins, a formalism plugin, so that different specification formalisms can be *contractually* handled, and an architecture plugin, so that a contracting kernel model can be applied to different kinds of architecture, mainly components and services. Taking into account formalisms is done through a DSL, that enables formalism experts to give to the system a generator of interpretation rules corresponding to specifications. The central model reifies them as contractual specifications, i.e. predicates following the assume-guarantee semantics [AL93]. This enables the system to compute responsibilities on specifications when they are interpreted as contracts. Moreover, two major properties can then be checked on contracts, i.e. *conformity* of a component against its specification and *compatibility* between specifications of horizontally or vertically composed components. On the architecture side, both architectural and behavioral observations must be defined in the framework so that contract construction and checking can be automated.
The implementation of our framework is currently based on the `Julia` implementation of the *Fractal* specification, like the *ConFract* system implementation. It actually reuse and extend this implementation so that i) architectural events needed by the framework are provided by some *Fractal* controllers, ii) contracts are built and updated by a contract controller interacting with other controllers, iii) contracts are checked by interpreting the rules on event triggering.
Three formalisms have been studied for integration, executable assertions with `CCL-J`, behavioral *CSP-like* specifications with Behavior Protocols, and sequence-based specifications with TLA [Lam94] (see also [COR06]). A combination of the first two have been used on a developed *Fractal* based client/server application that demonstrates this part of the integration as well as the architectural integration with hierarchical component-based systems. Nevertheless, *Fractal* is the only architecture that has been completely integrated in the framework. Demonstrations over service, or service and component based systems like SCA [Ope07], were missing. The following section 2.3 deals with the FAROS project and will show some reuse of the contracting kernel in a model driven toolchain targeting both service and component based systems.

2.3 From a Framework to a Model-Driven Toolchain

This section shares material with the CAL'08 paper "Vers l'intégration dynamique de contrats dans des architectures orientées services : une expérience applicative du modèle au code" [MBFCL08] and with several external deliverables of the ANR FAROS project [12]. It is therefore related to several collaborative works within the project, with Mireille Blay-Fornarino, Laurence Duchien, Philippe Lahire, Sébastien Mosser, Alain Ozanne and Nicolas Rivierre.
The FAROS project took place between 2005 and 2008. It aimed at defining a composition environment for building reliable SOAs (Service Oriented Architectures) for applications to be used in ubiquitous environments. The main approach is to use contracts between various stakeholders and

[12] http://www.lifl.fr/faros

entities of the application at the design level, in order to ensure a coherent composition of the services and components at the implementation level.

2.3.1 Motivations

In order to ease construction and maintenance of large and evolving information systems, SOA is usually presented as an architectural style allowing companies to transform existing resources into reusable and decoupled business services [MLM+06]. Building services from the composition of several services is made explicit by a workflow which result is a service itself.
Focus on business services and loose coupling between them are key benefits for high-level designers [Pap03]. To manage architectural benefits of SOA, fhe fusion of service-oriented and component-based architectural styles led to proposals such as SCA (Service and Component Architecture) [sta07]. However the integration and management of service-oriented applications and their necessary technical services, such as security, transactions, or QoS (Quality of Service), induced to take into account infrastructure and implementation targeted platforms. Many research work thus focused on the composition of services, including adaptation of composite services (process, workflow) at runtime [CM04], context-awareness of QoS measures for cooperating applications [BMFGI06], management of their constraints (Service Level Agreement) [BG05, ACD+05] and dynamic discovery of new services.
In this context, developers and administrators have to build and maintain highly scalable decoupled information systems, on a variety of platforms and constantly evolving technologies. Moreover They also have to reliable systems, which rely on and provide adapted forms of guarantees, from business SLAs to other contracts related to the software architectures or low-level QoS dimensions. There are advances and partial solutions in the different layers of the considered software systems, but major issues are the heterogeneity of execution platforms and the low-level abstractions offered to business architects. This problem requires that the deployment of service compositions and associated guarantees (performance, security, etc., specified at business level) are as much as possible automated.
This section shows how our contracting metamodel has been integrated as a central part of the FAROS generic process for building reliable Service-Oriented Architectures for ubiquitous applications. We mainly focus on the processing of contracts towards execution platforms. Besides, the *ConFract* contracting system was also used as one of the targeted runtime platforms. It is notably used to illustrate the projection from the abstract contract models to the platform specific models.

2.3.2 FAROS Process Overview

In order to ensure a coherent composition of services and components, the FAROS process is aiming at using contracts between different entities. The contract elements capture different properties, e.g. functional and non-functional, and are used throughout a model-driven composition environment to integrate contracts from business level to various service and component based platforms. As shown on figure 2.14, the process relies on three dedicated metamodels. High level abstractions, dedicated to business domains and independent of underlying technologies, are offered to business architects and captured by business metamodels (upper part of figure 2.14). A particular focus is placed on a central metamodel integrating the concepts of service composition, contractual guarantees and behavioral aspects. The diversity of the targeted infrastructures (orchestration or component) and specific contracting capabilities (policies, assertion language, aspects, etc.) are handled by platform models conforming to dedicated metamodels (lower part of figure 2.14). The central metamodel decouples business and platform levels and consequently, it allows transforming a business model

and its associated constraints into a pivot model, and later into an executable instance. Operations, such as validation, can then be applied on the central models regardless of the contexts of business or platforms.

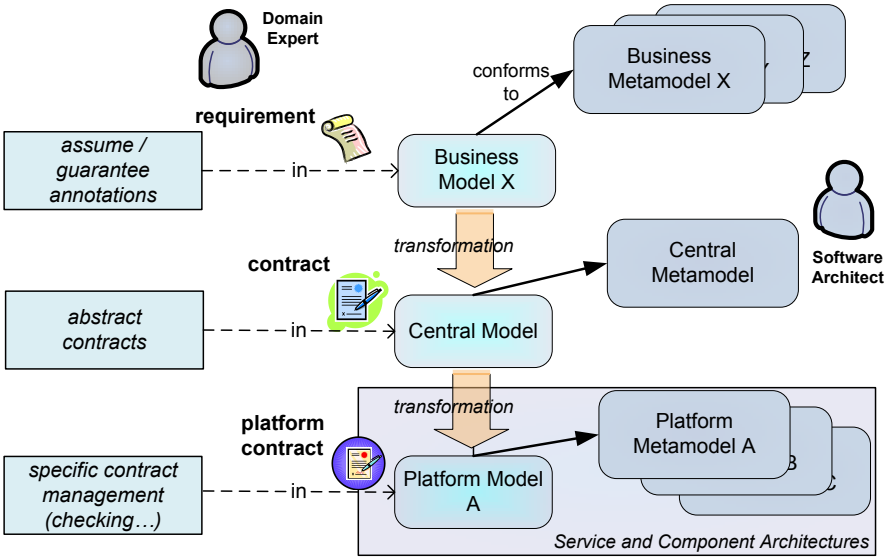


Figure 2.14: Models and metamodels in the FAROS process.

The transition between different models is handled by transformation. This approach helps to maximize the capitalization of know-how separately. The central metamodel also allows for reusing transformations, so that structured information is preserved and external tools such as checkers can be integrated during the development process. For instance, the central metamodel hosts the validation of service compositions and the processing that prepare projections to platforms through model transformations.
Regarding contracts, they are first expressed as constraints on the business models. These constraints are organized following the assume / guarantee semantics [AL93], so that they can be organized into contract models inside the central model. These contract models use a contract metamodel directly inspired from the Interact one presented in the previous section (see next paragraphs for details). Finally, these contracts are refined with *checker* model, and the transformation to the targeted execution platforms maps to a specific contracting runtime support, from basic checking code to first-class aspects or full-fledged contract objects.
The FAROS process has been developed to be generic enough to cover different service or component platforms. Platform metamodels have been designed for five different platforms:

- Adore [MBFR08], a web service orchestration engine that supports behavioral evolution with an uniform and merging-based model for orchestration and evolution. Two versions were developed: a first one in which each contract is then transformed in an Adore *fragment* to be then merged in the main business process, another one (Adore/Coconet [MBFCL08]) in which contract checking code is generated inside web service implementations.
- AoKell [SPDC06] is a aspect-oriented implementation of the *Fractal* component model. An extension of it was developed for FAROS and transformed contracts are aspects that are woven in components.

- ◇ ORQOS [BRL07], an extension of a BPEL engine to take into account static and dynamic QoS policies. Contracts are then transformed into ORQOS policies for events related to services.
- ◇ WComp [TLR+09], a lightweight service execution environment for ubiquitous event-based applications, which supports dynamic orchestrations. Contracts are then transformed into WComp specific aspects that are composed.
- ◇ *ConFract* (see section 2.1), which is supported by generating `CCL-J` specifications from the FAROS contracts, as illustrated below.

The approach is illustrated with a diversity of business domains (information broadcast in schools, personal medical record system, smart appliance for electrical networks). The process is tooled through EMF models and metamodels, and the transformations are implemented using the Kermeta[13] workbench.

2.3.3 Metamodels and Integration of Contracts

Figure 2.15 depicts the structural part of the central metamodel. It defines the organization of software

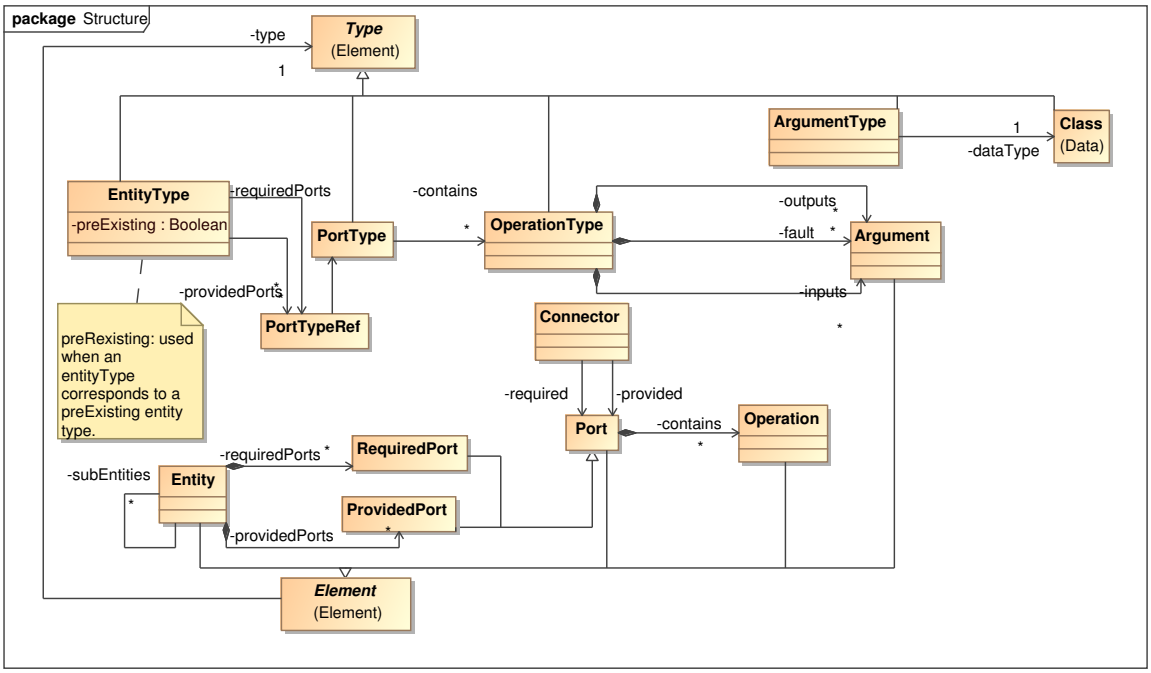


Figure 2.15: Structural part of the central metamodel (from FAROS deliverable F-2.3 [DBFC+08]).

Elements on which contracts can be applied. This metamodel is aimed at being sufficiently generic to consider the structural dimensions of a service oriented architecture or a component-based architecture. The concept of service or component is notably represented by an *Entity* and its type *EntityType*. There is a support in the metamodel for an *EntityType* being defined by the process, being pre-existing in the application, so that existing code can be represented when combined with generated one. An

[13] http://www.kermeta.org/

entity also provides *Operations* via a *ProvidedPort*, requires *Operations* through a *RequiredPort* and can have sub-parts.

Each structural *Element* of Figure 2.15 can be associated with one or several contracts, which representation is given in the metamodel of figure 2.16. The upper part of this figure is the representation of specifications organized as assumed/guarantee clauses and agreement between them, i.e. the kernel part of Interact presented in section 2.2.7. Thus *Contract* has participants (*Element*), is made of clauses which are defined by a *Specifications* attached to a type. A *Specification* associates an assumption (*assume* relationship) and a guarantee (*guarantee* relationship). Each clause references one of its participants as responsible for the clause, in terms of assume/guarantee logic. *ActionGuarantee* is simply the reification of an action to be triggered in case of violation.

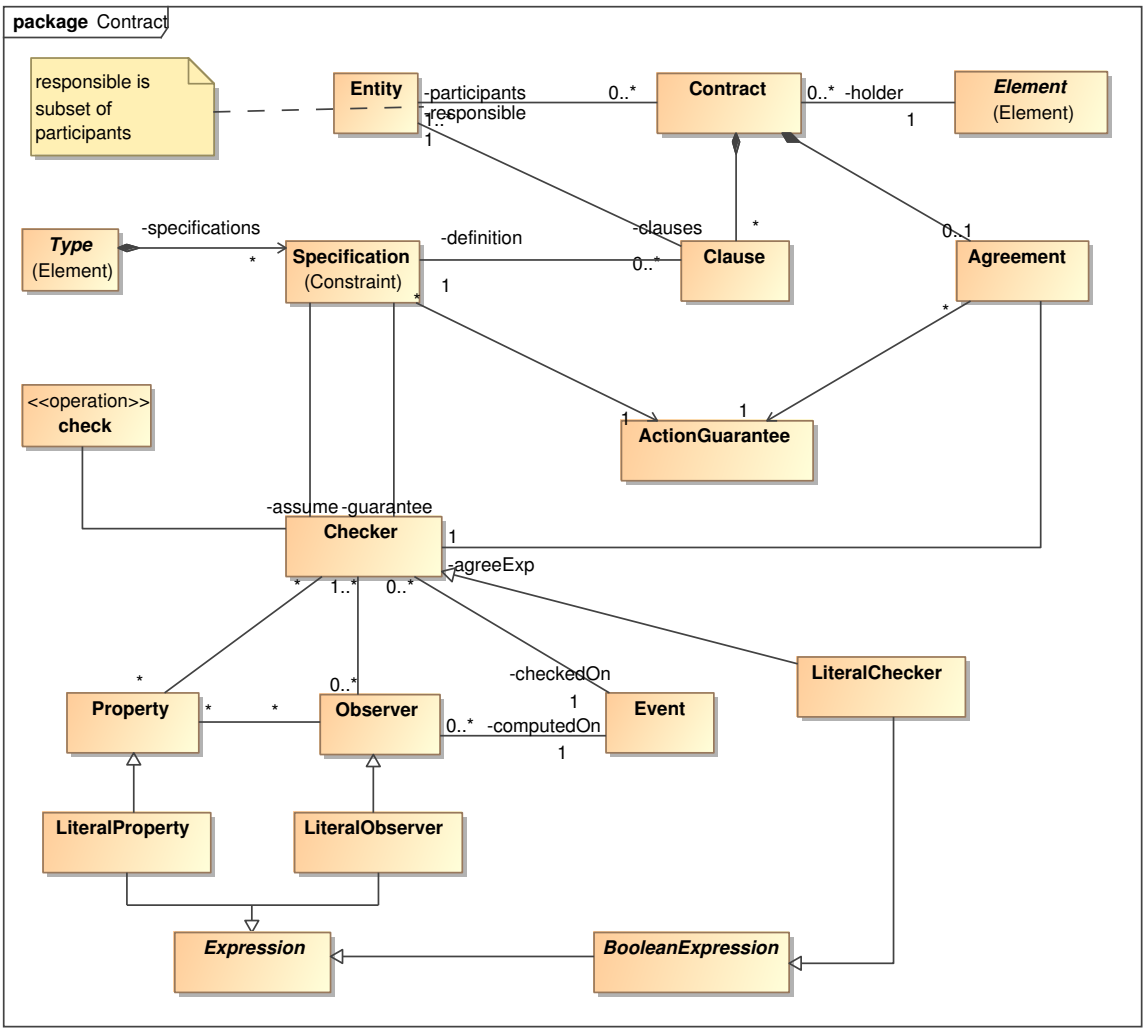


Figure 2.16: Complete contract metamodel (extracted from [DBFC+08]).

Specifications and agreements are carried out by a *Checker*, which is the main entry point to the part that reifies the checking process in the metamodel (lower part of figure 2.16. A *Checker* allows the registration of the activation events (*checkedOn*) that will trigger the checking operation (*check*). It

can get values from *Observers*, which are value containers that are computed on a specific *Event*. They characterize the elements of an application which provide the information in accordance with the context of a contract checking, i.e. any monitored value such as system time, the resolution of a screen of any complex values computed from the execution environment. There is also a support for *Property*, so that some non-functional properties that need to be made explicit at some points of the process — such properties can be explicit right from the business model or only visible in execution platform models —. The value computation of a property can also be implemented by an *Observer*, so that the event triggering this computation is made explicit. The *checker* finally checks its expression with all property and observer values. Events referenced in the contract metamodel are also organized in a metamodel not show here. This metamodel allows for representing life cycle events of entities (connections, start/stop...), communication (request sending/receiving, operation entry and exit...) and application specific events. Moreover checkers, observers and properties can be entirely defined in the model, and they are then *Literal...*, or they can be black-boxes as they represent checking mechanisms already defined in some execution platforms. Finally, these contracts are to be transformed to some model elements conforming to a targeted platform metamodel.

2.3.4 Illustration

To illustrate our approach, we choose an example taken from the FAROS validating case studies. We use timeout constraints, which are a typical requirement in numerous locations within information systems, either at the business or technical levels. For example, in information broadcast systems such as the SEDUITE or DMP applications (cf. section 2.3.2), a user will wait for its information in front of a public screen for at most an average maximum time. This timeout constraint should be propagated to technical layers of the software architecture in which information from various sources is dynamically aggregated in information data flow. There are also different means to verify this kind of contract and finally, if runtime checking is needed, there are different means to measure tome according to execution platforms: event-based systems can support time-stamping, orchestration engines need to be carefully crafted to take time before/after receiving/calling, etc. A possible deployment of the SEDUITE application is to broadcast information in colleges or universities, e.g., news and timetables of identified people are concatenated and combined to be displayed on several screens. The underlying business metamodel is thus concerned with the modeling of information sources composition and broadcasting on different devices, while a conforming business model would be defined for the specific application. The business expert would then add a predefined timeout constraint, parametrized by a delay, in the business model, so that the time to broadcast information on a screen is limited, as well as the time for a source to deliver its data.
Applying the transformation defined on the business metamodel to the central metamodel generates a `Timeout` contract model. Figure 2.17 shows a contract between a screen and an information provider. It specifies when does the timer start and stop, and the comparison between the computed delay and the set threshold. According to the constrained elements, e.g. required or provided port, the checking moment differs. This is captured in the different observers and the checker, which refers to events *GetInfoOperationEntry* (for *BeforeTime*) and *getInfoOperationExit* (for *AfterTime* and the *LiteralTimeChecker*). In the central model, we do not completely express how the contract is checked. It is in the transformation to the platform model that implementation details will be made explicit, i.e. how to determine the time, while preserving the observations and checks coordination according to events. Nevertheless, literal checkers are used to analyze the data flow to detect inconsistencies, such as services requiring delay from called entities shorter than the delays they guarantee or more compositional properties on the sum of all used service timeouts against a global timeout threshold.

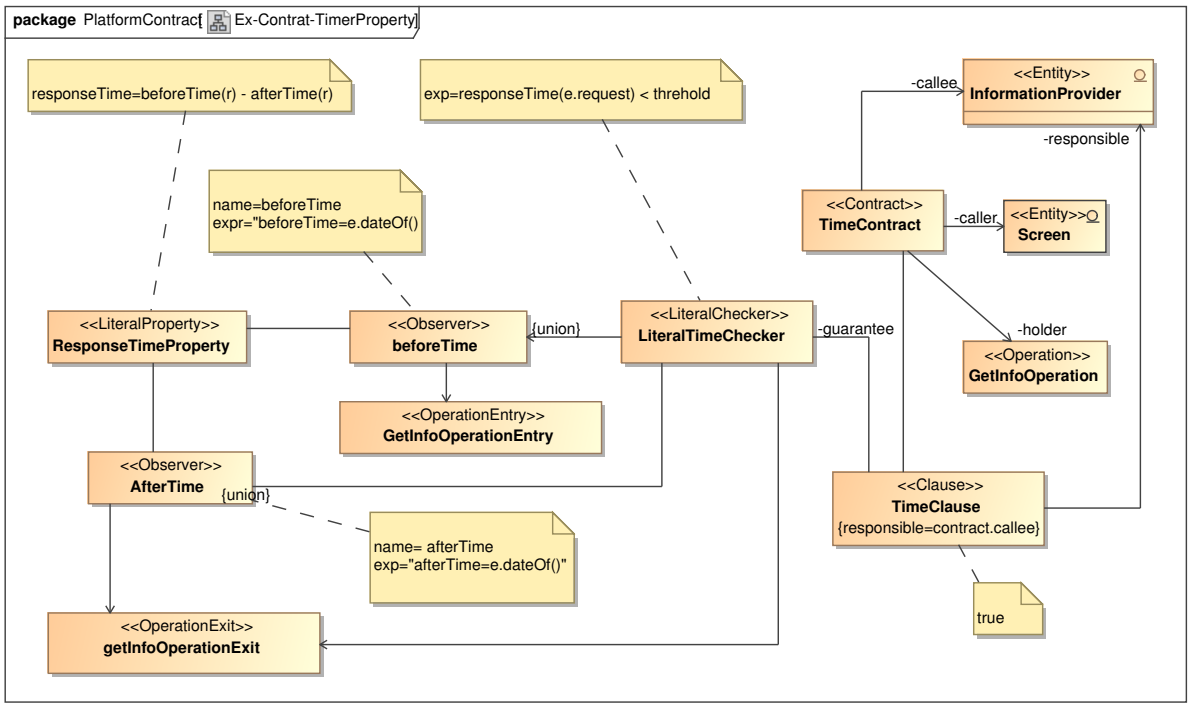


Figure 2.17: TimeoutContract on InformationProvider (extracted from [DBFC+08]).

As platform metamodel, we use the *CCL-J* metamodel, which has been created in the FAROS project to cover one possible of platforms, i.e. a language based platform. Other FAROS platform metamodels cover creations of aspects, dedicated components, etc. Figure 2.18 gives an overview of the *CCL-J* metamodel. This metamodel describes the kinds of specification through a class hierarchy and the structural elements of a *CCL-J* specification as associated concepts. Each specification form thus inherits from *CCLJSpec*, relies on a *Context*, which is an operation for pre and post, and an interface for invariants. A specification can also contains an assertion with its label and its expression. Taking our timeout contract example, figure 2.19 illustrates what classes and information are transformed to obtain a *CCL-J* model The contract scope is used to determine the context of the specification (*OperationContext*). The clause and its responsibilities (client/supplier relationship) drive the transformation towards a interface specification (*InterfaceCCLJSpec*). The analysis of the *LiteralTimeChecker* and of its two observers leads to a postcondition (*Post*), which will generate the final expression. It is through the detection of a pattern among observers and events (values computed on OperationEntry and reused on OperationExit) that the expression is generated. Here the properties are not kept on the targeted platform, so only their observation part is used. Constant values like *threshold* are generated in an utility class to be easily modified.
The generated model can then produce the following textual specification in *CCL-J*:

```
context Info i.getInfo()
  post: System.nanoTime() - System.nanoTime()@pre
                          < FAROS_constants.timer_threshold;
```

This specification can then serve as input for the *ConFract* system, generated contract object at configuration and execution times (cf. section 2.1).

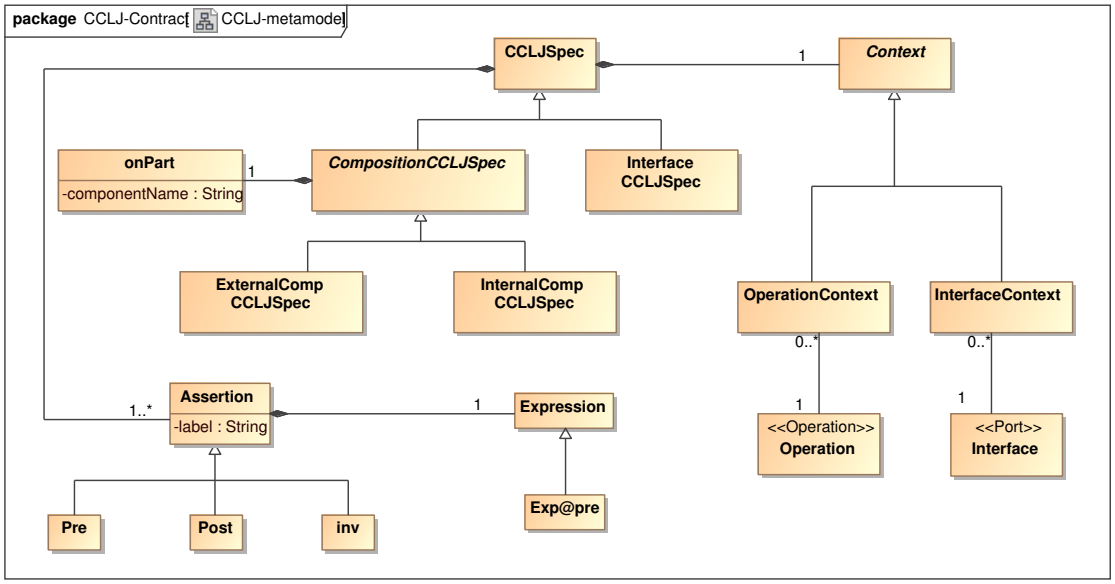


Figure 2.18: CCL-J metamodel (extracted from [DBFC+08]).

2.3.5 Discussion

We only showed above the projection towards the *ConFract* system, but the targeted platforms in the FAROS project cover different forms of contract implementation, ranging from programs (Adore/-Coconet), to aspect technologies (AOKell, Adore, Wcomp) and language (*ConFract*, ORQOS). The abstract contract models capture and distinguish moments that determine the evaluation context of contracts from moments they are checked, but the construction of the context is very different among platforms and some expressions or elements in contract models cannot be generated on them. For example, when projecting to Adore, *fragments*, a.k.a concerns to be merged in the orchestration, are generated. The transformation encapsulates some mapping between kinds of contracts, e.g. a time-out, and a fragment template. This template has parameters that define the way to relate a contract to a service, e.g. when a contract needs to check something at operation entry, it will be placed before the activity that is used for merging fragments, so that the checking will be done before the service execution.

Two other kinds of contracts have also been experienced on all platforms: an authentication contract checking that a client calling a service has already been authenticated through another dedicated service, and a architectural contract checking whether a given component/service is present. Studying the code generation needed on targeted platforms, we experienced the difficulties to specify at the model level and to automate the support for checking while ensuring that constraints will have a equivalent semantics between model and execution platforms. This is obviously due to the weak, but open, semantics that is underlying the event-based description of checkers in the central metamodel. Nevertheless, we acknowledged the fact that there exist *contract patterns* that are difficult to generate, but can be easily provided by the platforms. We identified some possible patterns with the developed examples, i.e. timeouts, data and access constraints, architectural constraints, call history... These patterns were sketched during the last stages of the FAROS project and the lack of time did not allow for developing further this concept. This is mainly due to the large and unforeseen effort deployed to

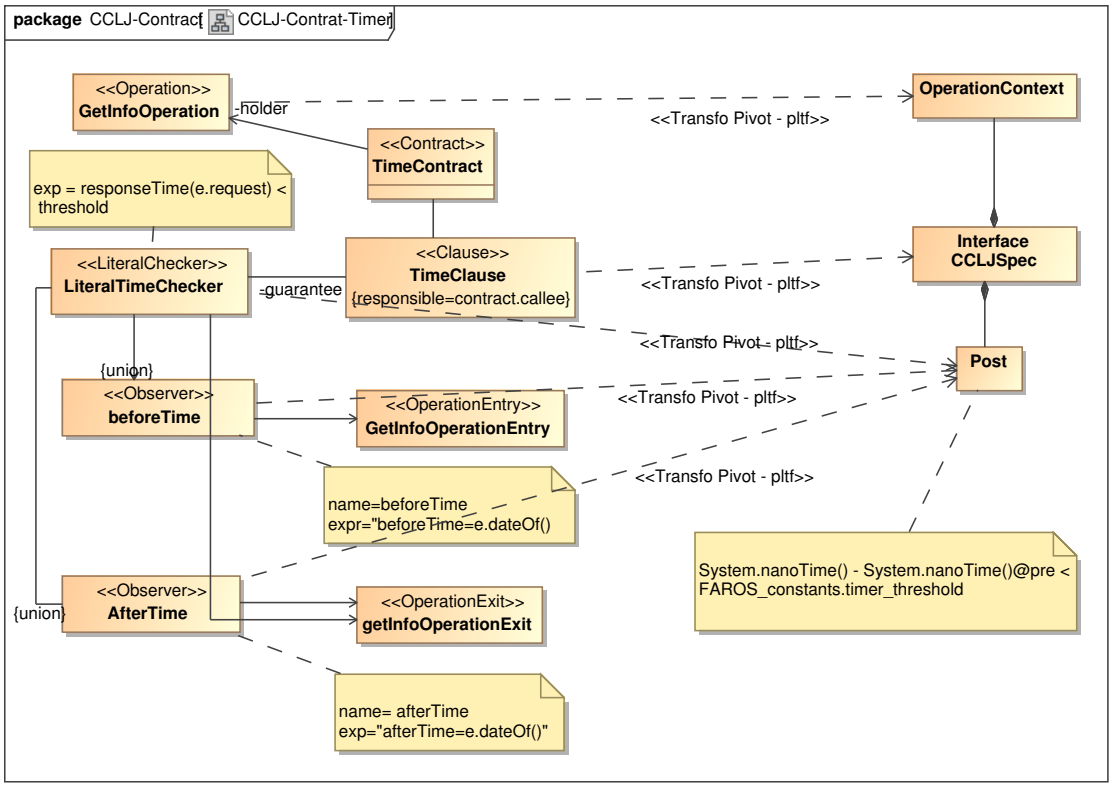


Figure 2.19: Transformation of the Timer contract into *CCL-J* model (extracted from [DBFC+08]).

make the end-to-end toolchain handle service and component architectures, an example of accidental complexity of current MDE techniques [FR07].

2.3.6 Summary

We described some results of the ANR FAROS project which aimed at providing a MDE composition workbench for building reliable service and component architectures with the help of an end-to-end contracting support. In the FAROS process, the software architectures and the contracts are captured at three different levels, with metamodels at each level. Business metamodels are described and corresponding models are annotated with constraints. A central metamodel decouples business and platform levels, and integrate concepts of service composition, contractual guarantees and checking. Platform metamodels capture the various targeted execution platforms.

In this project, our contribution is related to the provision of the central contracting metamodel, which is directly inspired by the Interact one presented in the previous section. The metamodel is also largely extended to capture observation and checking operations and model them in connection with an event model. These events allows for defining how contracts should be checked while being open to interpretation on the execution platforms when they are transformed towards them. Our *ConFract* system was also one of the targeted platforms and we showed how the central contracts were transformed into *CCL-J* specification models.

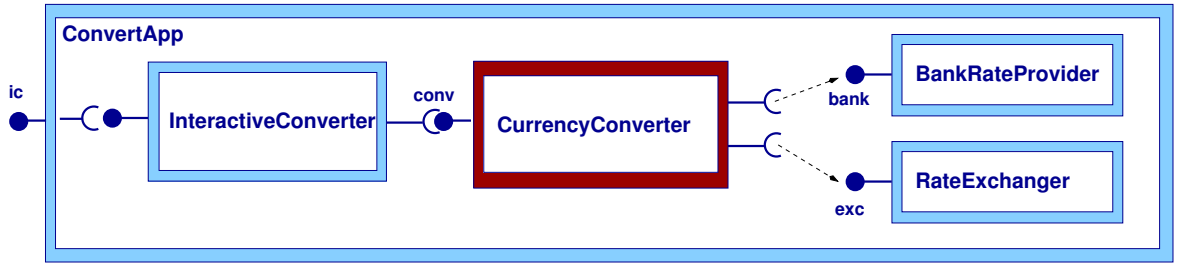
2.4 Contract-based Self-testable Components

This section shares material with the Fractal CBSE Workshop at ECOOP'06 paper "Specification of a Contract Based Built-In Test Framework for Fractal" [DC06]. It concerns a collaborative work with Daniel Deveaux.
Continuing our line of research around software contracting, we now describe some work that aims at providing a testing framework in which rich components become self-testable through the addition of our software contracts and associated built-in test suites.

2.4.1 Motivations

With the emergence of richer component models, e.g. *Fractal*, Koala, SOFA, etc. [LW07], the discipline of CBSE has made significant improvements. Such models support forms of horizontal (connections between components through explicit provided and required ports) and vertical (composite components compositions. If component models have been more powerful, less effort has been put on providing some testing architectures that correspond to the need of both unit and integration testing of these richer forms of component. Considering their features, the approach of *contract-based testing* [Mad03] seems particularly suited, as the notion of contract is a fundamental part of the definition of software components [Szy02] and the approach itself is now considered as one of the best testing techniques [Bin96]. This comes from the ability of contracts to be used at the same time as a software documentation and design approach [Mey92] and a support for class testing, using the responsibilities they set between classes to determine salient oracles.
At the end of the nineties, an approach of *"Contract-Based Testing"* was proposed by several authors [Mad03] and in CBSE, contracts have been also studied for component design [BJPW99, Szy02]. At the time of this work, several proposals had been made to construct contract-based built-in tests (CBBT), as a methodological approach [Gro05] or with practical frameworks [VFH05]. In [BAM03] the authors propose an UML based approach to incorporate built-in tests into Java components through aspect-oriented techniques. Components are equipped with an additional testing interface, as defined by [AG02], and this interface provides state-related information of the component. The contract is established between the component and its client through the interface. A component is thus a black-box and if it has some required parts, they are not taken into account in the testing framework. Aspect-oriented techniques are used to easily produce component implementations in a Java platform. Some other implementation framework relies on executable assertions to base testing on behavioral contracts [VFH05], or on some timing constraints to determine the fulfillment of a component's response time requirements when it operates in a client-server relation with other components [GMR05]. We see in all these approaches a common point in the fact that they consider contracts on a client/server mode very similar to object-orientation. Components are tested through one of their provided interfaces, but nothing is provided to support testing with required interfaces connected to some forms of stubs or real components.
In this work, our objective was two-fold. First, we aimed at providing a contract-based testing framework that fully exploits the rich features of available component models, taking *Fractal* and *ConFract* as starting points. Second, we wanted to provide a support for the concept of *self-testability* [DFJ01] adapted to the same software components. This approach has been developed in the STclass system, a contract-based testing framework for Java [DFJ01]. A Self-Testable class supports a so-called *"Design for Trustabilty"* [TDJ99] (DfT) process The goal is to maintain high quality along the life cycle of library classes. The DfT uses a *"test first"* approach like eXtreme Programming or JUnit, but with a strong structuring of the specifications using contracts: the tests are defined before the code

implementation, but not really first, because they are built from the contracts which capture specifications. Using STclass tool, the benefits of built-in testing for object-oriented languages have been demonstrated: tests are not limited by the encapsulation and built-in tests can be replayed in various situations, in particular for regression testing. The framework allows the inheritance of contracts and tests, supporting the reuse and the uniqueness of the validation. Another important result shared by STclass and JUnit concerns the management of test results. For regression analysis and maintenance, it is very important to store on the long term all test results in a structured reusable form and to have sophisticated statistics and report tools.



Interfaces Signature

```
public interface ExchangeRates {
  // list of managed currencies
  public Set currencies();
  //get the exchange rate for a specific conversion
  public float exchangeRate(String inCurr, String outCurr);
  // get the decimal count for a specific conversion
  public int decimalCount(String inCurr,String outCurr);
  // is this currency managed by the module
  public boolean isValidCurrency(String curr);
}
```

```
// from JUnit tutorial
public interface Money {
  public Money add(Money m);
  public boolean isZero();
  public Money multiply(float factor);
  public Money negate();
  public Money substract(Money m);
  public float amount();
  public String currency();
  public String toString();
}
```

```
public interface Converter {
  // make the raw conversion without bank rate.
  public Money convert(Money m, String toCurrency);
  // make the conversion applying a bank rate.
  public Money convertThruBank(Money m, String toCurrency, BankInfo
      .TransferType tt);
}
```

```
public interface BankInfo {
  public enum TransferType {CREDIT_CARD,CASH,INTERBANK};
  // get the bank rate for the bank 'bankName' and transfert type '
      tt'
  public float rate(String bankName, TransferType tt);
}
```

Figure 2.20: CurrencyConverter architecture.

2.4.2 Illustration

To illustrate our contribution, we use a simplified currency conversion application made with *Fractal* components (see figure 2.20). It can be seen as an extension of the example used in the famous JUnit tutorial [14] [GB99]. An equivalent `Money` interface is notably used in this application.

[14] `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`

The application is implemented as a composite component, `ConvertApp`, which contains four subcomponents. `InteractiveConverter` is connected to the interface (`m` of type `Main`) provided by `ConvertApp`. It controls interactively the conversion required by the user. `CurrencyConverter` is the component that effectively realizes the conversion through its interface `conv` of type `Converter`, but it also needs two external functionalities, designed as two required interfaces. The first one, `bank` of type `BankInfo` enables it to obtain the transfer rate of a bank according to the kind of transaction (by card, in cash, etc.), while the other one, `exc` of type `ExchangeRates` is used to obtain supported currencies, their exchange rates and the number of decimals needed for an accurate conversion. `BankRateProvider` provides the interbank rates and `RateExchanger` provides the functionality of currency exchange mentioned above.
We now focus on some of the possible constraints that can complement the architecture of the application, write the corresponding *CCL-J* specification and shows the resulting contract in *ConFract* (see section 2.1 for details on *CCL-J* and *ConFract*). First, on the `ExchangeRates` interface, the resulting conversion rate, which must be a positive float number, is obtained from two valid currency names (strings). Moreover, on the same interface, assuming that currency names are valid, the number of decimals for a conversion is also obtained, and must be between 1 and 5.

```
context float ExchangeRates.exchangeRate(String inCurr, String outCurr)
    // spec1
  pre: isValidCurrency(inCurr);
  pre: isValidCurrency(outCurr);
  post: result > 0.0;

context int ExchangeRates.decimalCount(String inCurr, String outCurr)
    // spec2
  pre: isValidCurrency(inCurr);
  pre: isValidCurrency(outCurr);
  post: 1 < result < 5
```

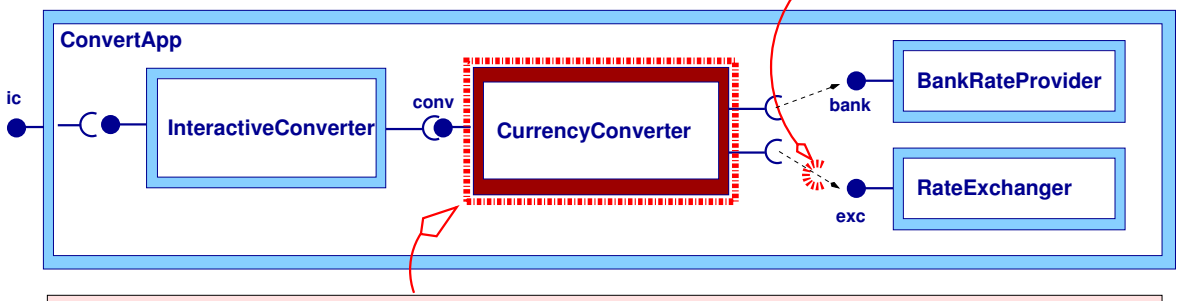
This results in an *interface contract* generated by the ConFract system, as shown on top of figure 2.21.
A more complex specification can be defined on the main conversion method. The conversion of an amount `m` from a currency to another (`toCurrency`) with a transfer rate `tt` is made by the method `convertThruBank`; this method can be executed if the source and target currencies are valid – obtained through the required interface `exc` –; the method must then return a `Money` instance with the targeted currency, and the backward conversion of its value must be equal, with some decimal approximation, to `m` minus the *bank commission* – obtained on the other required interface `bank` – :

```
on <CurrencyConverter>                    // spec3
  context Money conv.convertThruBank(Money m, String toCurrency,
      TransferType tt)
    pre: exc.isValidCurrency(toCurrency);
    pre: exc.isValidCurrency(m.currency());
    post: result.currency().equals(toCurrency);
    post: MathUtil.approxEqual(result.amount(), convert(m, toCurrency).
        amount() * (1 - bank.rate(<this>.attr.bankName, tt)), exc.
        decimalCount(m.currency(),toCurrency)-1);
```

The above specification refers to three interfaces, `conv`, `bank` and `exc`, and produces an external composition contract (see bottom of figure 2.21). The responsibility model associated to the external composition contract takes into account the hierarchical nature of the component assemblies. The surrounding component (`ConvertApp`) is responsible of the *compositional preconditions*, as it is

the only component that can act on its subcomponents so that they all together ensure the exposed property. The beneficiary of these preconditions and the guarantor of postconditions is the specified component.

```
interface contract on <ConvertApp>   // pretty print of the contract object
server: ExchangeRates <RateExchanger>.exc
client: ExchangeRates <CurrencyConverter>.exc
participants: <RateExchanger>, <CurrencyConverter>
clauses:
  float exchangeRate(String inCurr, String outCurr)      // spec1
    pre    guarantor: <CurrencyConverter>    beneficiary: <RateExchanger>
      isValidCurrency(inCurr); isValidCurrency(outCurr);
    post   guarantor: <RateExchanger>    beneficiary: <CurrencyConverter>
      result > 0.0;
... // spec 2, etc.
```




```
external composition contract on <CurrencyConverter> // pretty print...
participants: <CurrencyConverter>, <ConvertApp>, <InteractiveConverter>,
                    <BankRateProvider>, <RateExchanger>
clauses on server interface Converter conv :
  Money conv.convertThruBank(Money m, String toCurrency, TransferType tt) // spec3
    pre     guarantor: <ConvertApp>   beneficiary: <CurrencyConverter>
      exc.isValidCurrency(toCurrency);
      exc.isValidCurrency(m.currency());
    post   guarantor: <CurrencyConverter>   beneficiary: <InteractiveConverter>
      result.currency().equals(toCurrency);
      MathUtil.approxEqual...                          // remainder of spec 3
    ... // other clauses, other interfaces, etc.
```

Figure 2.21: Resulting contracts on the CurrencyConverter.

2.4.3 Testing Framework Overview

The testing framework follows three simple principles, which are applicable to hierarchical component models in general.

Writing tests only relies on basic concepts. In order to write and run tests, testers should only handle basic concepts in the component model. As in JUnit or in STclass [DFJ01], in which tests are defined as simple DSLs, testers should only use components and small configuration scripts to define and run tests. This leads to using components as testing elements such as drivers and stubs, thus

providing an uniform view of the component and its testing artifacts. On the contrary, the framework should manage all the complexity associated to test automation (management of test suites, results collection, etc.).

Testing is strongly contract-based. Postconditions and invariants in contracts make good and salient oracles. Few extra oracles have to be defined and testing units code consists only in simple method calls on interfaces. In addition, preconditions limit the scope of the test. It is then not necessary to test situations that are always rejected by contracts. Contracts that share the execution sequence in very small units make possible a clear identification of responsibilities when a fault is detected. Like in STclass [DFJ01], another benefit of using contracts is to split into two separate descriptions the specification and the test scenario.

Tests are built-in. The component to be tested should contain, or have reference to, all information relative to its own testing, making it *self-testable*. This consists of information to generate the provided interfaces and test drivers, information to generate testing stubs and storage of its own test results.

Besides, to capitalize on OO unit testing habits, the concepts of TestUnit, TestCase and TestSuite are reused with a similar semantics. These concepts are organized as in STclass [DFJ01], which is close to JUnit with some minor differences:

- ◇ ***TestUnit*** defines a scenario, with a segment of (Java or other) code, which is applied to one or more provided interfaces of the *"Component Under Test"* (CUT).
- ◇ ***TestCase*** defines a test environment for the TestUnits. It is able to create, initialize and connect testing stubs and to define local testing ressources; it contains five definitions: i) an *architectural setup* that defines the stubs' connection and configuration, ii) *data setup* actions to configure states of both the environment and the CUT before testing, iii) the *list of UnitTest*s that can be activated in this environment, iv) a *data teardown* actions to reconstruct the initial testing environment (mainly external data used or modified), v) an *architectural teardown* that restore the initial isolation. The first three definitions are mandatory, whereas both teardown actions can be omitted — if no architectural teardown is defined, a default isolation will be realized by the framework.
- ◇ ***TestSuite*** is a simple ordered list of TestCases, TestSuites or TestUnits (following the traditional composite pattern) that can be activated during the test.

With this organization, the same TestUnit can be used in several TestCases and can be activated (following the TestCase definitions) for different kinds of test. These concepts will be illustrated in the following paragraphs.

2.4.4 Supported Testing Modes

The framework aims at supporting all functional testing situations all along the component life-cycle.

Black-box testing in isolation. First the component can be **tested in isolation**, usually during its design and implementation. Considering the whole CUT as a black box, its external composition contract controls the dependencies between its interfaces, and the interface contract on its provided interface gives the functional oracles for the test. In our currency converter example, the external composition contract enables us to build a first unit test corresponding to the general case. It is going to call the method `convertThruBank` with 1 euro for an interbank conversion in dollar. The environment has thus to create an object Money that has a value of 1 euro:

```
Money oneEuro = new MoneyImpl(1.0,"EUR");
```

and the test simply calls the method:

```
conv.convertThruBank(oneEuro, "USD",BankInfo.INTERBANK);
```

The test oracle is then the external composition contract of the figure 2.21. As a result, the tester does not have to create result objects or to define explicit oracles, since postconditions ensure that the currency and the value of the result are correct.
Figure 2.22 illustrates the architecture and test elements that are necessary to perform this test. The two components `BankRateprovider` and `RateExchanger` are disconnected and respectively replaced by two stubs: the `BankStub` component can only return a fixed transaction rate, while respecting the interface contract). It could be more sophisticated, using for example a mechanism of configuration by attribute so that setup code of test cases can configure it. As for the `RateSub` component, it can be simply implemented as a conversion euro to dollar and incrementally improved afterward with other conversions, as new test cases are added.
A second TestUnit (`convert_02`) illustrates another capability of the framework, i.e. the explicit definition of an oracle when interface contracts are not sufficient to verify the oracle, often when the test composes actions. In our example, it consists in checking that the backward conversion is correct. From these two first tests, one may consider further tests grouped in the same test case for the method `convertThruBank`. These other tests would use other amounts, other currencies, and the two other forms of conversion.

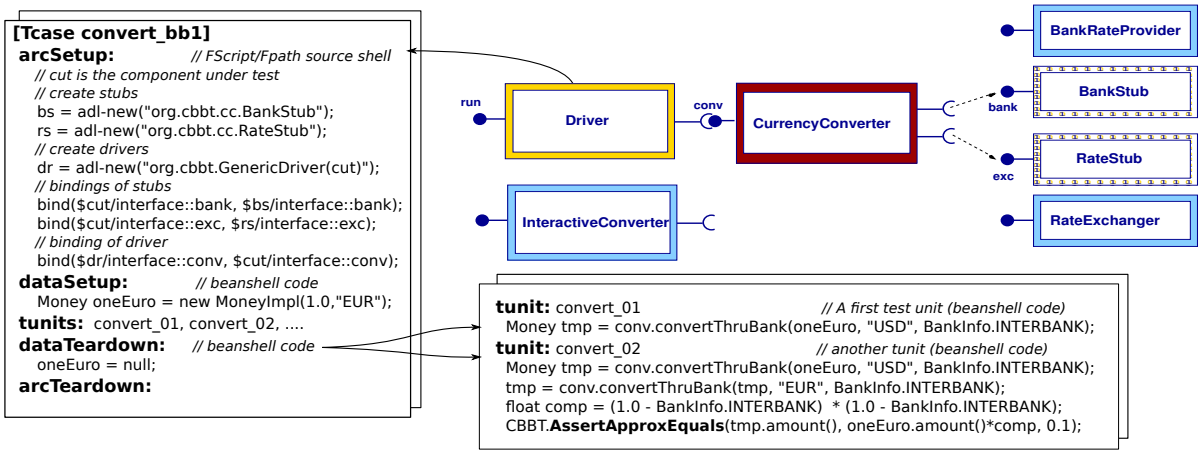


Figure 2.22: Black-box testing in isolation.

Figure 2.22 also shows the definition of TestUnits and TestCases, in separate syntactic (files) units, as a TestUnit can be used in several TestCases. As for test cases, it must be noted that the architectural setup and teardown are defined using an extension of the Fscript language [DLLC09]. This DSL relies on a notation inspired by the XPath language for XML, so that navigations and dynamic reconfigurations of Fractal architectures can be performed through simple an readable queries. On our example, the architectural setup uses some Fscript operations to create stubs and drivers and to bind them to the CUT. Bindings use the navigation to get the appropriate interfaces from the concerned components. Our testing framework extends the language by providing some default values for `cut` as a reference to the tested component, as well as for any component connected to it before testing.

The data setup and teardown are using Java code as scripts, which are interpreted by a beanshell interpreter (cf. section 2.4.5). The code inside TestUnits is also Java scripts, with the addition of some variables to access the provided interfaces of the CUT (`conv` in our example). The language used above is just an example, but it demonstrate clearly that the tester manipulates only simple declarative language elements. All operations specific to component implementations are handled by the testing framework. Finally, it should be noted that the preconditions filter out any test that would not use known currencies. A violation of preconditions shows that the CUT is not to blame, but that its environment, test driver or stub, is badly configured.

In situ testing. A second step is related to the integration of the CUT with its effective providers. This implies two activities: i) *requirement testing* which verifies that some external provider component, intended to be connected to a required interface of the CUT, respects the contracts on the connected interface; ii) *integration testing* where the CUT, connected with its actual suppliers, is tested on its provided interfaces, using both interface and external composition contracts as oracles. It should be noted that, for both of these forms of testing, tests should be able to be replayed in case of dynamic reconfiguration. Actually an *in situ* test realizes the integration testing of the CUT with its actual providers, it is then crucial that this test can be also run during or just after some reconfiguration actions. The requirement test is also particularly useful if the external component is a Component Off The Shelf (COTS), which has neither contracts nor built-in testing equipment. As shown on figure 2.23, to realize this test, a specific test driver is connected to the external provider component. As in black-box testing, it will be driven by TestUnits and TestCases, but in the `arc_setup` part, no stubs are used and the driver is not connected to the CUT. The `arcSetup` only consists in a direct connection between the driver and the tested provider component.

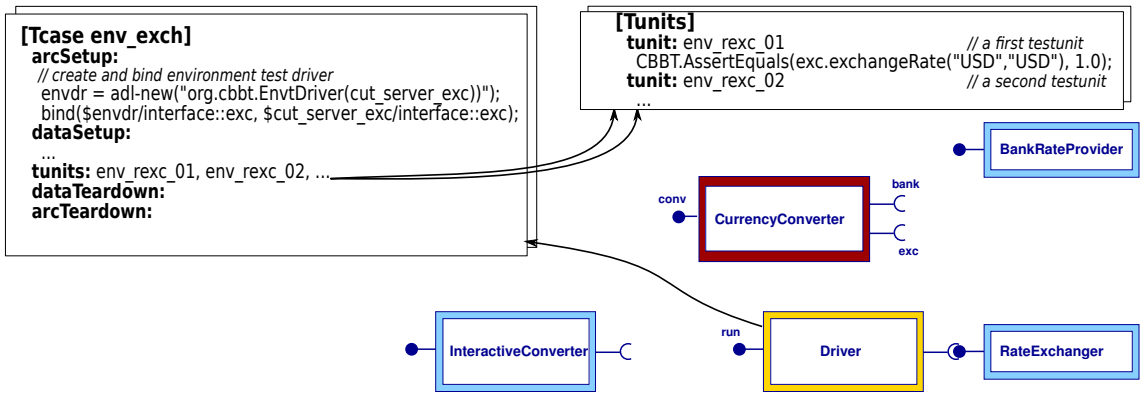


Figure 2.23: Requirement testing.

As for the integration test, it is also very similar to the black-box testing, but the CUT is directly connected to its real suppliers rather than to stubs, as shown on figure 2.24. Obviously the integration can be progressive and a test may simultaneously use stubs and real suppliers (cf. figure **??**). As a result, the description of the integration for the developer is very similar to a black-box test description, only the `arcSetup` (and eventually `arcTeardown`) differs, the TestUnits can be the same.

Gray-box testing. The framework also supports a form of gray-box testing, that is some testing on the content of the tested component. During the black-box testing of a component, enabling the

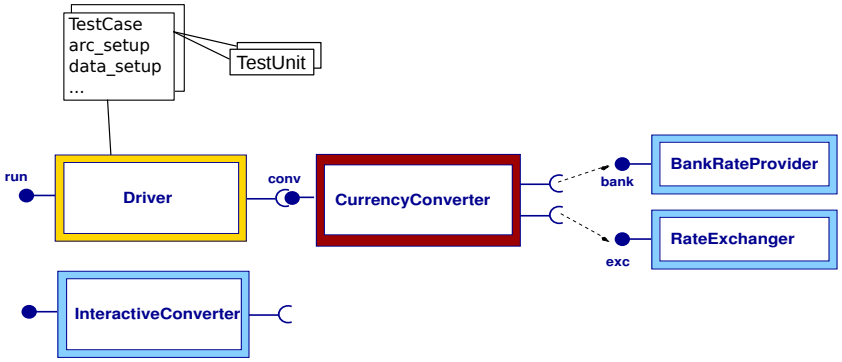


Figure 2.24: Integration testing.

verification of its internal composition contract turns the process into a gray-box testing, in which its *implementation* is tested. The same elements of the framework are then used in this situation. As the considered component model is recursive, its implementation can be some code or an assembly of components, with their own contracts. It is also possible to imagine hierarchical testing of the inner side of the component, but this aspect is out of the scope of our contribution.

2.4.5 Test Management and Framework Implementation

Using the proposed framework, tests and contracts can be considered as built-in, like in physical electronic components. Each component has a reference to the description of its contracts and tests and a block of management code, which i) interprets test and contract descriptions, ii) creates the driver and stub component, iii) runs the tests and iv) collects test results. An obvious way to integrate this technical feature into *Fractal* components is to equip each component with a specific non-functional interface, a *test controller*, through which an architect or some runtime code can control testing.
To become *self-testable* a component should contain or reference all information relative to its own testing. The testing controller is thus able to get contract information from the contract controller (see section 2.1.6) and to get test units, cases and suites as separated files organized in the file hierarchy, just like files for Fractal ADL and Java implementation of primitive components.
This *testing controller* also acts as a generic runner for all tests related to its component and manages the history of all test results. As An important facet of the test activity is the test reporting and archiving, test results of the framework are stored in XML report files that can be processed to automatically generate many forms of test reports and which allow for easy long-term archiving.
When a test running method is called, the testing controller first isolates its component by unbinding both its required and provided interfaces (the component should have been stopped first). It then creates a surrounding component, called the testbed, so that the component is put inside and other components needed for testing are also created inside. According to the arguments of the running method, it creates test drivers connected to provided interfaces of the component under test, testing stubs connected to its required interfaces and/or *bypass* binding to some actual provider components. In case of integration testing, only test drivers connected to the actual provider component are created. While the type of test driver components is dependent from the provided interfaces under test[15], their implementation uses reflective capabilities to be generic. It consists in an interpretation engine,

[15] the Fractal type system does not allow a component to change its type, e.g. to add or remove one of its interfaces.

based on the Beanshell[16] Java scripting language, which i) calls the architectural setup in Fscript, ii) interprets the data setup in Java, and iii) interprets the test units, using also reflection to call the methods on the provided interfaces.
Before running the test cases, the testing controller needs to be notified of any contract violations on the tested parts of the component. It thus registers itself to two contract controllers: the one of the surrounding test bed (for the external composition contract and interface contracts of the CUT) and the one of the component under test (for its internal composition contract).

2.4.6 Summary

We gave an overview of a testing framework that exploits our *ConFract* contracting system so that targeted components become *self-testable* through the addition of built-in test suites that exploit contracts as oracles. We show how our testing framework enables developers and architects to organize tests in different ways. Black-box testing can be made in isolation with specific component acting as drivers and stubs around the component under test. Tests are organized as units, cases and suites, but are complemented with specific forms of setup, both for the architecture and for the data. Built-in tests can also concern the environment of a deployed component, testing its providers, and they can also be used for integration testing when the component is connected to its actual providers at assembly time. According to the management policy of dynamic adaptations, the same tests as in the assembly phase can be passed again after some dynamic reconfigurations. The reflexive capabilities of *Fractal* makes this possible as runtime components can be entirely reconfigured so that the framework is able to manage dynamic connections and tests. The framework makes also possible to perform contract checking and testing at runtime for critical software systems. In this case a resulting overhead will appear and an interesting open issue is the definition of a *self-test supervisor* that could schedule components self-tests, managing the whole system efficiency and workload.

[16] `www.beanshell.org`

CHAPTER 3

Self-Adaptation

Contents

3.1 Negotiation of Non-functional Contracts 54
3.1.1 Motivations 54
3.1.2 Case Study 55
3.1.3 Negotiation Model 56
3.1.4 Concession-based Policy 58
3.1.5 Effort-based Policy 60
3.1.6 Negotiable Contracts in Autonomic Control Loops 62
3.1.7 Implementation and Self-Adaptive Capabilities 63
3.1.8 Related Work 65
3.1.9 Summary 65
3.2 Compositional Patterns of Non-Functional Properties 66
3.2.1 Objective 66
3.2.2 Classification of Non-Functional Properties 66
3.2.3 Modeling Patterns 68
3.2.4 Reasoning Support for Compositional Properties 69
3.2.5 Mapping Patterns to the Fractal Platform 71
3.2.6 Illustration 72
3.2.7 Exploitation in Contract Negotiation 73
3.2.8 Implementation 77
3.2.9 Related Work 79
3.2.10 Summary 80
3.3 Self-adaptive QoI-aware Monitoring 80
3.3.1 Objective 80
3.3.2 ADAMO Underlying Model 81
3.3.3 ADAMO Architecture 83
3.3.4 Elements of the Framework 84
3.3.5 Implementation 86
3.3.6 Patterns and Extension Points 86
3.3.7 Self-Adaptive Capability 87
3.3.8 Related Work 89
3.3.9 Summary 89

This chapter presents our research work on providing self-adaptive capabilities to contracting and monitoring systems. This research was conducted from 2004 to 2010 and is now continued by ongoing work on architecting self-adaptive systems (see chapter 5).
Self-adaptive capabilities are provided by software systems to cope with changes at run-time [LRS00]. The relevance of engineering self-adaptive capabilities is mainly due to the continuous evolution from software-intensive systems to ultra-large scale systems [NFG+06]. This means that the self-adaptive capabilities should support and facilitate run-time decisions to control structure and behavior of the system. With our background on providing contract support in software intensive systems, we were interested in relating software contracts to self-adaptive capabilities, going beyond the symptoms of changes that a contract violation represents. This led to research on negotiation mechanisms for component contracts. As monitoring is clearly the common activity between contract checking and adaptive systems, we also focused on providing self-adaptive monitoring systems with advanced functionalities.

3.1 Negotiation of Non-functional Contracts

This section shares material with the Euromicro-SEAA'05 paper "Fine-grained Contract Negotiation for Hierarchical Software Components" [CC05], and the ATC'06 paper "From Components to Autonomic Elements using Negotiable Contracts" [CCOR06]. It is related to a part of Hervé Chang's PhD Thesis and a collaboration contract with France Télécom R&D (now Orange labs).
Our work related to contracting started on the observation that contracts are a well-adapted means to organize stable properties on software entities, enforcing and reusing constraints on their interactions and non functional aspects. Nevertheless, providing powerful contracting systems is not an easy task with strong requirements both on software systems, such as 24/7 operation implying dynamic reconfiguration capabilities, and on the internal architecture itself using hierarchical components for the sake of generality. Among the advances that we have made, the capability to manipulate contracts as first-class entities at configuration and run times is particularly relevant. This form of model at runtime [BBF09] allows for automatically checking again some properties of a software architecture after it has been dynamically reconfigured. But on the other hand, with highly fluctuating non functional properties and different kinds of changes dynamically happening, contracts are frequently challenged and violated.

3.1.1 Motivations

Contract violations correspond to relevant changes in the system behavior, that could be exploited to modify the system itself. If these adaptation operations are embedded in the software system and automatically executed, it naturally becomes a form of *self-adaptive software*, partly following the definition of [LRS00]: "it evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible". Handling basically the contract violations necessitates to catch the corresponding exception and attach some adaptation code. With many contracts that may be violated, *ad hoc* code is spread out on the architecture, hampering maintenance. There is thus a need to find the appropriate mechanisms and tools to make a trade-off between reliability, ensuring stable properties with contracts, and flexibility, with dynamic adaptations on contracted components.
In component-based systems, some mechanisms handle the runtime fluctuations in QoS properties and the available resources using a monitoring system combined with basic adaptation rules which

perform component re-parametrization or structural modifications [DC01, DL03]. When they define a notion of contract, it is restricted to the architecture of the system, ensuring correct connections between components [DC01]. Other middleware platforms provide QoS control and measurement capabilities of particular critical network-related properties through reflective and adaptive techniques [BAB+00, N. 01] which directly provide integrated control mechanisms without explicit representation of non-functional properties. Other techniques consist in selecting other implementations or QoS profiles that fulfill the specification [GPA+04a], or in trying statically-defined alternate services requiring lower quality levels [LS04]. These works use a notion of contract to organize the knowledge of different functioning levels, but the negotiation mechanisms are not made explicit in the runtime architecture, i.e. components are not empowered with the right to negotiate on properties they are responsible for. Being concerned about automatically restoring, in most cases, the validity of contracts, we proposed negotiation mechanisms, inspired from those conceived in multi-agent systems, which make it possible to adapt components or contracts at configuration and run times, with the aim to restore the validity of established contracts.
Besides, as numerous challenges are directly related to the software engineering of self-adaptive systems with feedback control loops (find an appropriate control model [HDPT04], architecting the loop [KPGV03a, GSS04, LPH04a]...), we also studied the interpretation of the resulting negotiable contracts in feedback loops following the *Monitor-Analyze-Plan-Execute-Knowledge* (MAPE-K) architecture [KC03, IBM01].
The contribution here relies on a general negotiation model, in which contract violations are handled by activating a negotiation process for each violated clause of a contract. A notion of negotiation policy enables one to finely control the negotiation process. Integrating the negotiation mechanisms in the *Fractal* reflective component model also allows for making theses mechanisms self-adaptive, notably to avoid some harmful behaviors.

3.1.2 Case Study

As an illustrative example, we reuse the application and contracts on the *Fractal* multimedia player described in section 2.1.2. Figure 3.1 shows the surrounding `FractalPlayer` component with its five connected subcomponents managing GUI, core playing, video configuration, battery probing and logging. It also depicts the textual form of an external composition contract objects built with the *ConFract* system. This contract is related to the `start` method of the *Fractal* interface named `mpl`, ensuring that the video can be played (precondition using the `Configurator`) and that the played url is part of the `History` at the end (postcondition). In the precondition the `VideoConfigurator` component evaluates, through the `canPlay` method, the `Player` ability to entirely play the given video, considering different parameters such as system parameters (e.g., battery level) and the video source (taking into account the decoding complexity of the video). Figure 3.1 shows also the responsibilities computed in the contract, i.e. *guarantor*, *beneficiary*, and *contributors* (participating component being necessary to check the contract). Details can be found in section 2.1.5. Responsibilities in our example of external composition contract can be summarized as follows:

Interface role	Construct	Guarantor	Beneficiaries
server (mpl)	pre	<fp>	<pl>
server (mpl)	post	<pl>	<fp>, <gl>
client (c)	pre	<pl>	<fp>, <vc>
client (c)	post	<fp>	<vc>

Once built, contracts may be challenged by the fluctuations in extra-functional properties, such as variations in available resources, changing requirements and environment, or by the dynamic recon-

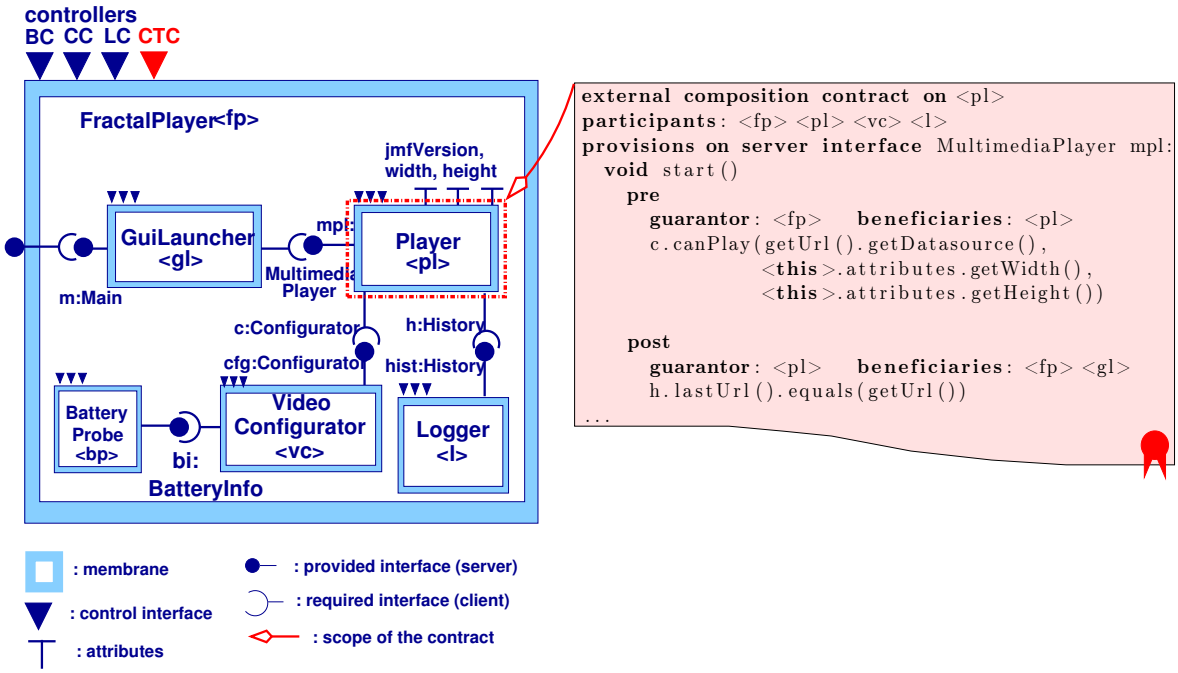


Figure 3.1: External composition contract on the *Fractal* multimedia player.

figurations of components. By default, the *ConFract* system handles contract violations by notifying the guarantor component with the violation context. In an *ad hoc* approach, it is then possible to manage these violations by adding specific handling code throughout the architecture.
A better organized solution consists in reusing the responsibilities of each clause (provisions in the contract object) as they precisely determined all responsible components. Putting negotiation capabilities inside components allows for involving them in an automatic negotiation process which aims at restoring the validity of contracts. This negotiation can occur either at assembly time or at run time. At assembly time, such negotiation assists the system integrator to detect and to solve incompatible constraints before execution, and to leverage the qualities of assemblies while taking into account each component specification. At execution time, this negotiation contributes more to increase the autonomy of the application by handling contract violations in an automatic way. In our example, the negotiation could then lead to lower battery consumption by reducing the video display size, or to completely withdraw the constraint, thus implying that the video might be stopped if battery gets weak.

3.1.3 Negotiation Model

The negotiation model aims at restoring the validity of violated contracts by making the responsible components negotiate the violated contracts in which they are involved. As contracts are composed of provisions, negotiation processes are done at the granularity of each violated provision of a contract. In the *ConFract* system, contract provisions both specify functional and extra-functional aspects. The negotiation of functional aspects is more appropriate during testing. On the other hand, as extra-functional aspects address configuration and run time qualities of components (usually classified under the larger concept of Quality Of Service) and their relationship with the environment (deployment

constraints), we mainly focus on negotiating provisions which specify such aspects.
For each violated provision, an *atomic negotiation* is thus automatically built and activated. Each atomic negotiation precisely defines the *negotiation parties* and *protocol*. Accordingly, given each participating component role, it is possible to develop different negotiation *policies* which define the participants reasoning and drive the negotiation. We developed a *concession-based* and an *effort-based* negotiation policies, that take advantage of the various component responsibilities.

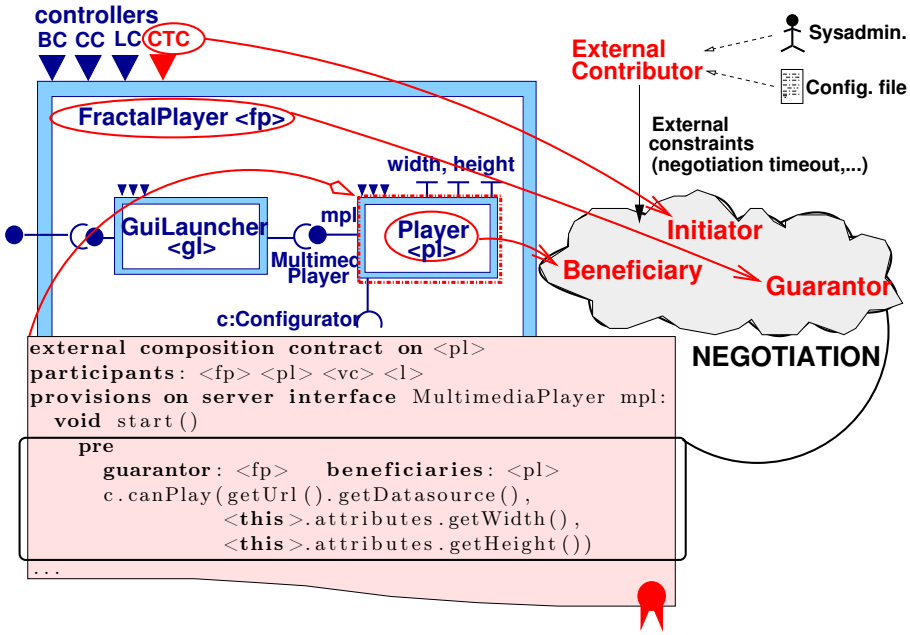


Figure 3.2: Negotiating parties for the precondition of the external composition contract.

An atomic negotiation involves *negotiating parties* and follows a *negotiation protocol* partly inspired from the extended *Contract-Net Protocol* (CNP) [Smi80]. This protocol, commonly applied in multi-agent systems for decentralized tasks allocation, basically organizes the interactions between a manager and contractors following three steps: announcing, bidding and rewarding. In our model, we retrieve this organization by defining as negotiating parties *(i)* the contract controller in the role of the negotiation initiator, which controls the negotiation process, as it manages contract life cycle and operates contract checking, *(ii)* participants, which are composed of the participants of the provision, and of an *external contributor* which helps representing interests from a "third party" with deeper decision ability. For example, this external contributor could be the system integrator willing to inject higher level constraints into the system (e.g deployment constraints) or various data to parametrize the negotiation process (e.g the negotiability of the provision [17] given a specific deployment context, negotiation timeout, propagation of negotiation information to lower hierarchy levels).
In our example that refers to the external composition contract, the negotiating parties of the precondition on the `start` method are the contract controller of `<fp>` as initiator, `<fp>` itself as guarantor and `<pl>` as beneficiary (cf. figure 3.2). For the postcondition, the parties are the contract controller of `<fp>` as initiator, `<pl>` as guarantor, and `<fp>` and `<gl>` as beneficiaries.
The negotiation initiator and the various participants thus interact following three steps (request of proposals, proposal of modifications and re-checking of the provision against the proposed modifica-

[17] A provision is *negotiable* if the negotiating parties agree to negotiate it.

tion), and the responsibilities of participating components are exploited to develop the two following negotiation policies.

3.1.4 Concession-based Policy

The *concession-based* policy provides a first kind of negotiation behavior which is based on a process of concession-making. By taking advantage of their type of responsibility, the negotiation initiator thus interacts with the beneficiaries components, and it requests from them to rely on an under-constrained clause. These beneficiaries may then propose some concession proposals which can only refer to the negotiated provision object. Such proposals can lead to change the whole provision or some of its parametrized elements in the same current execution context, e.g. some function parameters, or to completely withdraw the provision.
In order to completely define the concession-based policy, the beneficiary role is refined by introducing a *principal beneficiary* and a *secondary beneficiary*. Principal beneficiaries are directly concerned with a clause as they have the ability to act during the negotiation process. On the opposite, secondary beneficiaries are quite passive and cannot make the negotiation progress. They can only validate the proposed changes. In our example concerning the postcondition of the `start` method of the `mpl` interface, `<fp>` is the principal beneficiary because it is responsible of the correct usage of the contract (played video history is correct) and can act on its subcomponents during the negotiation. `<gl>` is the secondary beneficiary because it only appears as a simple client of the video player service and does not have knowledge of the other interfaces referenced in the contract. Using an extension of the responsibility model, the contracting mechanism automatically detects principal from secondary beneficiaries.
When the verification of a provision fails, the concession-based negotiation process is decomposed into three steps, as described in Fig. 3.3.

1. in step 1, the initiator requests the negotiability of the violated clause from the beneficiaries and it evaluates the overall negotiability by computing a weighted linear additive scoring function.

2. in step 2, if the provision is negotiable, the initiator requests concession proposals from principal beneficiaries and for each proposal, it performs changes on the provision and re-checks the provision. If a proposal re-validates the provision, the atomic negotiation is successfully completed and changes are committed.

3. in step 3, if proposed changes are not satisfactory or the withdrawal of the provision has been issued[18], the initiator asks to principal and secondary beneficiaries for permission to withdraw the provision.

To successfully act during the negotiation process, the decision model of the principal beneficiaries is based on sets of alternatives which express their preferences. Thus, a component named C can define the following set of alternatives:

$$\mathcal{A}_{\#p,C} := \{A^1_{\#p,C}, A^2_{\#p,C}, ..., A^n_{\#p,C}, \text{STOP or RELEASE}\}$$

to negotiate the provision identified as $\#p$. For every concession proposal requested by the initiator, the component C will successively propose its preferred alternative among this set. In this policy, an alternative $A^i_{\#p,C}$ corresponds either to provision or some of its parametrized elements, and STOP

[18] A proposal can consists in the *withdrawal of a provision* to suggest to completely remove the provision.

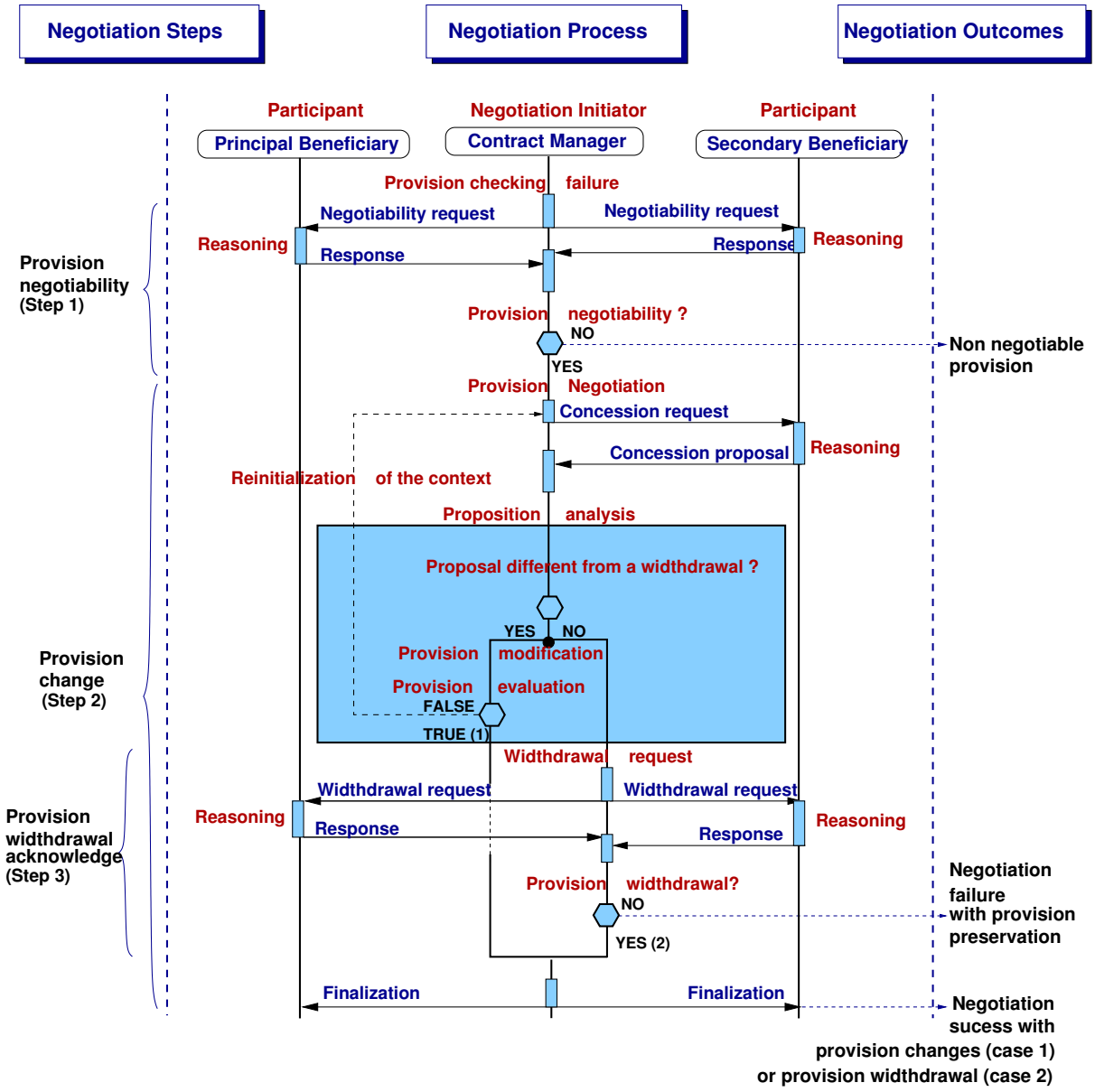


Figure 3.3: Concession-based negotiation process.

or RELEASE are used to notify the end of the concession process while retaining the provision or withdrawing it.
In our example, the provision that ensures the ability to entirely play a video can only be checked at run time since it requires the execution context and up-to-date resource information. If the verification of this provision fails, the concession-based negotiation process would involve the contract controller of <fp> as the initiator and <pl> as the unique and principal beneficiary. The negotiation outcomes may lead to progressively reduce the video display size in order to decrease battery consumption. If these concessions are not satisfactory, the provision could then be completely withdrawn, and in this case, since the constraint has been discarded, the video might be interrupted if the battery level becomes weak.
In this scenario, <pl>'s successive concessions may be driven by the following set of alternatives:

$$\mathcal{A}_{pre,<pl>} := \{(width \leftarrow \frac{width}{\sqrt{2}}, height \leftarrow \frac{height}{\sqrt{2}}), \text{RELEASE}\}$$

With this set of alternatives, <pl> initially proposes an alternative describing the changes on its parameters `width` and `height`. The initiator performs the change and evaluates the provision. If the verification succeeds, the negotiation outcome is successful, otherwise the initiator cancels the changes, and requests for a new concession to which <pl> responds by proposing the provision withdrawal with the RELEASE alternative. The negotiation finally terminates with the provision withdrawal since <pl> is the only beneficiary.
In the same way, the clause concerning the correct history of played videos can only be checked at run time. The negotiating parties would be the contract controller of <fp> as initiator, <pl> as guarantor and <fp> and <gl> respectively as principal and secondary beneficiaries. During the negotiation process, <fp>, with the set of alternatives $\mathcal{A}_{post,<pl>}$:={RELEASE}, would propose the provision withdrawal and the initiator would consult <fp> and <gl> to definitely decide to withdraw it.

3.1.5 Effort-based Policy

The *effort-based* negotiation policy is proposed to enrich the model with different kinds of negotiation. To restore the violated clause, this policy focus on exploiting the responsibility of the guarantor which has to ensure the provision. The overall negotiation process using the effort-based policy follows the three steps defined in the negotiation protocol. However, it differs by the negotiation proposals issued by the guarantor component. The negotiation initiator requests *efforts* from the guarantor which can propose two kinds of efforts according to its responsibility of either, the implementation of the terms referred in the negotiated provision or, the assembly of its subcomponents. In the first case, the guarantor is responsible of the implementation of the terms in the provision, and it can act to restore the clause by doing some action efforts at its level. In the second case, the guarantor does not implement the terms of the clause, but some of its subcomponents does. Therefore, it is now responsible of the assembly of its subcomponents and may act by propagating the negotiation process down its hierarchy, in order to consult the subcomponents which may propose efforts to restore the violated clause at the higher level.
The next section 3.2 further details the negotiation process for action and propagation proposals, based on patterns on the architecture.
Compared to the concession-based policy which deals with relaxing the negotiated clause, the efforts proposed by the guarantor component consists in doing some actions that aims at restoring the violated clause by changing its evaluation context. These actions can consist in changing the values of the terms referred in the clause, or executing an adaptation function that may perform any adaptation

code, provided that it keeps the negotiated contract clause unchanged and is realized under the scope of the guarantor component.

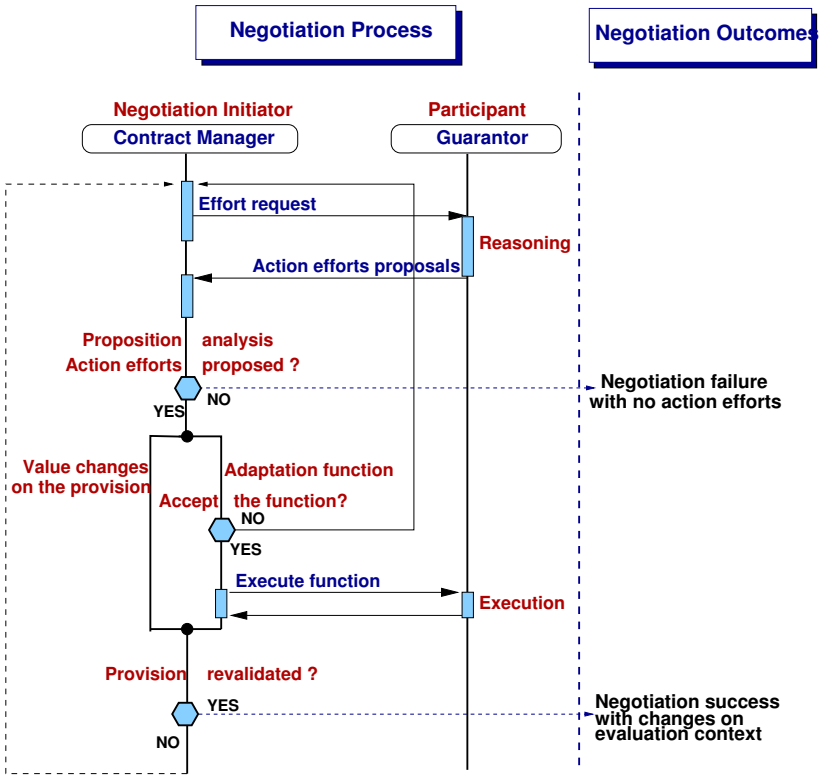


Figure 3.4: Action proposals in the effort-based negotiation policy.

Fig. 3.4 depicts the interactions between the negotiation initiator and the guarantor component. The first step which evaluates the negotiability is omitted in the figure as it is basically the same as in the concession-based policy. We suppose the clause is negotiable and in this case, the interaction steps are the following.

1. the initiator requests efforts from the guarantor.

2. if the proposed action consists in changing the values of terms referred in the clause then the initiator performs the changes and re-checks the clause.

3. otherwise, if it consists in an adaptation function, as this function remains private to the guarantor, the initiator can then only accept or refuse the execution of the function without knowing its details. If it accepts, the adaptation function is performed, and the initiator then re-checks the clause to assess its validity. Otherwise, if the initiator refuses, it simply requests for other action proposals.

4. in both two cases, the atomic negotiation is successfully completed, as soon as a proposal re-validates the clause, and it terminates with a failure, if none of the actions efforts restore the validity of the clause.

3.1.6 Negotiable Contracts in Autonomic Control Loops

As negotiable contracts intuitively provide some elements relevant to self-adaptive systems construction, we now briefly discuss the role of contracts in feedback control loop following the MAPE-K decomposition [IBM01]. In our system, contracts play a central role. They specify collaboration properties between parts of the system, they are monitored all along the life cycle of the system and updated according to architectural changes, and they provide a support on which the negotiation model relies to activate and finalize each atomic negotiation. In fact, contracts serve as knowledge and analysis tools to identify, in a fine-grained way, some part of the system to be monitored, the responsible components for each violated provision, and whether the proposed modifications revalidate the contract.
As contracts are managed by contract controllers in *ConFract* (see section 2.1), each contract controller provides support for instrumenting contracted components it is in charge of and monitoring them both at assembly and execution times. Then it checks the provisions, detects architectural or behavioral contract violations, and finally drives the negotiation activity according to the negotiation rules and policy. Therefore contract controllers are involved in each step of the control feedback loop. At any level of composition, a feedback control loop can use the different types of contracts (*interface contracts*, *internal and external composition contracts*) supported by *ConFract*. External composition contracts, for example, express the usage and external behavior rules of a component. They are well suited to develop a negotiation policy between a component and its environment.

- ◇ **Monitor**. Contracts define the spatial domains of the system that are visible, that is a component scope, and the temporal context under which provisions have to be checked. The provisions of a contract mainly describe where the observation occur in terms of parts of the system (external or internal side of components, interfaces), when to operate the checking rules, the values to capture, and the verifications to be made. The `CCL-J` language currently supported in *ConFract* and based on executable assertions (see section 2.1), allows one to easily perform sanity checks and detect contract violations by testing the validity of the input and output values of components.

- ◇ **Analyze**. Contracts represent a source of knowledge and can be exploited to shift towards finer understanding and analysis of the problems that occur in the system. They can be used to obtain various information that concern the locations of the system where the violations appear and the responsibilities of the impacted components. Each provision of a contract precisely identifies, among the set of participating components, the responsible components in terms of guarantor, beneficiary or contributor of the provision.

- ◇ **Plan**. To adjust the system in reaction against contract violations, atomic negotiations are activated and they organize the recovery process through a collaborative activity between the negotiating parties. Atomic negotiations are themselves organized through various *negotiation policies*. Such negotiation policies rely on the role of the participating components and define a complete negotiation behavior. Each proposed negotiation actions can be seen as part of a plan, even if this is a very simple instantiation of it, in comparison with planning and decision making approaches.

- ◇ **Execute**. The negotiation aims at restoring the validity of contracts. To achieve this, the actions to execute depend on the negotiation policy and on the own capabilities of the negotiating parties. These actions span from basic re-configurations — through attributes — of components

involved in the negotiation to more sophisticated modifications of the contract themselves, as well as advanced architectural changes. Finally, it should be noted that contracts are checked to validate the negotiation, so that they participate in ensuring the correctness of the adaptive actions of the feedback control loop.

During negotiations, components can be seen as configuring themselves when they try to (re-)establish valid contracts. A form of self-configuration is thus supported and negotiable contracts are intrinsically acting for the robustness of the overall system by helping each component to deal with detected failures. The following section on the implementation of our negotiation mechanisms will also show the limitations of the approach, as if contracts can be used with relevance in control loops, they are not sufficient to build complete self-adaptive systems.

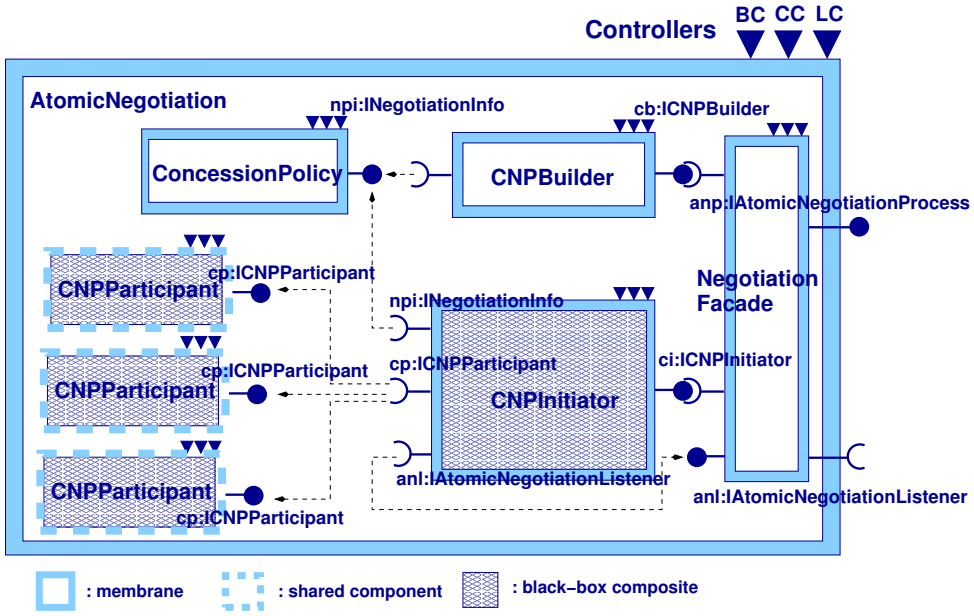
3.1.7 Implementation and Self-Adaptive Capabilities

The implementation of the negotiation mechanisms is realized upon the first implementation of the contracting system *ConFract*. Tt is based on the reference implementation in Java of the *Fractal* component model, named `Julia`. Negotiation mechanisms represent new control functionalities which are used by the contract manager. The negotiation model is itself implemented as full-fledged components which are created and hosted in the membrane of components, through dynamic reconfigurations of component membranes[19]. Although it would have been possible to implement the negotiation model using plain Java objects, we decided to use full-fledged components to explicitly represent the main components of the model as well as their architecture in a uniform approach that applies component-based structure and separation of concerns. With such an implementation, we also consider applying some forms of contracting at the (meta-)level of the negotiation system itself.
When the checking of a contract provision fails, an atomic negotiation composite component is dynamically built to represent the atomic negotiation process. Fig. 3.5 depicts the architecture of the atomic negotiation component. It encapsulates some proxy components to the various parties that have to negotiate and also other meta-components which are used to build and drive the negotiation process. To instantiate this atomic negotiation component, references to the responsible components are retrieved from the violated contract provision object according to the negotiation policy, and for each negotiating party (contract controller and responsible components), a proxy component which will negotiate on behalf of each party, is dynamically built using dedicated factories. These proxy components are composite components which notably encapsulate the negotiation reasoning of each corresponding negotiating party and they expose clearly defined interfaces, which then allows them to interact according to their role of initiator or participant in the Contract-Net protocol. The negotiation reasoning component of each negotiating party is instantiated from strategies defined in external XML files, and they can also be redefined at runtime, outside any atomic negotiation.
Using the component-based architecture of the controllers and the negotiation processes, some self-adaptive capabilities have been implemented within the negotiation mechanisms:

- ◇ Each negotiation action has a contract over its realization, so to detect timeout when interacting with each participant. This timeout value can be relaxed once, then an effort action is executed to forbid negotiation with the faulty participant.

- ◇ Each negotiation process has a contract over its realization, so to detect overall timeouts of an atomic negotiation. As default values for negotiation, this other timeout value can be relaxed

[19] While we implemented the first *ConFract* system, the `Julia` implementation added the capacity to implement component membranes (i.e. controllers) using objects but also components.



Legend: The atomic negotiation is a composite component that contains the main components required to negotiate (policy, initiator, participants) and utility components required to dynamically built the component.

Figure 3.5: Component-based architecture of an atomic negotiation component.

with two other values

◇ The two timeout negotiations described above are themselves controlled for timeouts but with a simple reactions that consists in deactivating all timeout contracts.

Additionally, we implemented an history of all negotiations, centralized on each *Fractal* runtime, so that oscillations on some contracts can be detected. Each negotiation, with its clause and results, is saved, so that cases of oscillation between similar provisions or functioning parameters used in negotiation can be detected. Oscillations can be experimented in our negotiation systems because we also added some other control loops that try to reestablish nominal values on negotiated contracts on a regular basis.

This clearly shows both the capabilities and the limitations of the negotiation mechanisms. On the one hand, the component-based organization of the negotiation mechanisms and processes allows for their own adaptivity through the very same mechanisms. On the other hand, the provided mechanisms presented here are not powerful enough to cover a complete self-healing mechanism. The main missing feature is the possibility to consider feedback control loops that are quite implicit to be completely explicit [BDG+09], so that they can be more easily organized and that some appropriate control model can be integrated when possible. Even with some smarter policies that could mix concessions and efforts to provide a form of regulation, the resulting regulation would still be subject to possible oscillations if the values exchanged during the negotiation create an overshoot in the control [HDPT04]. Currently, it is only possible to detect this overshoot and to deactivate the involved negotiation.

3.1.8 Related Work

Negotiation is used in multi-agent systems (MAS) to coordinate actions, resolve conflicts and above all share scarce resources. Several negotiation mechanisms base their interactions model on the general *Contract-Net Protocol* [Smi80]. The negotiation strategies are often specialized and dedicated to a specific application whereas, Faratin et al. defined in [FSJB99] advanced strategies with responsive and deliberative mechanisms. In our model, although we adapted the CNP protocol, the negotiation differs by dealing with the provisions of a violated contract and our strategies are defined by sets of alternatives closely related to work from Balogh et al. [BLH00]. Furthermore, while MAS are endowed with social and environmental autonomy, our approach is more closely related to self-adaptive systems and *autonomic computing*, by aiming at automatically restore the validity of contracts through adaptation of components or contracts at assembly and run times.

Numerous works use *QML* (*QoS Modeling Language*) [FK98a] to specify and contractualise QoS properties. In *QML*, extra-functional aspects are described by specifying expected quality levels. However, specifications in *QML* express relatively high-level constraints without explicit representation at run time. Particularly, it is not possible, contrary to *CCL-J*, to combine both functional and extra-functional aspects within a specification, as well as referring to component, interface or method parameters at the application level. *QML* is usually used to specify QoS in distributed systems [BG99, KS98] and component models [RBUW03, LS04]. In distributed systems, negotiation protocols deal with a restricted number of QoS parameters relating to network characteristics and mainly consider the negotiation as the process of reserving resources [SLM98, PLS+00]. In component models, most QoS negotiation protocols are dedicated to multimedia applications and the negotiation consists in controlling components admission and performing ressources reservation to support the agreement. In [GPA+04b], the negotiation protocol consists in selecting either an implementation or a QoS profile that fulfills the specification whereas in [LS04], negotiation is statically defined and consists in trying alternate services with lower quality levels. Our approach differs as we view the negotiation as a generic process of consulting and adapting responsible components or contracts to restore the validity of contracts.

3.1.9 Summary

We described mechanisms to support contract negotiation on hierarchical software components. These mechanisms were designed for *ConFract*. In the proposed negotiation model, the protocol is partly inspired from the extended *Contract-Net Protocol*, frequently used in multi-agent systems. A negotiation initiator thus consults clearly identified responsible components in order to restore the validity of violated contracts. Given component responsibilities, two complementary negotiation policies have been proposed. A concession-based negotiation policy enables beneficiary components to relax violated constraints. An effort-based policy exploits the guarantor component capabilities so that it can realize adaptation. The next section will detail an extension of this effort based policy so that compositional patterns of non-functional properties are exploited to propagate efforts among the relevant part of the component hierarchy.

The proposed mechanisms have been developed over the *ConFract* system and validated on a large component-based applications, Amui, that provides dynamic grouping capabilities on an instant messaging server, using the multimedia player used as example as one of the applications that is automatically launched between members of the same dynamic group.

3.2 Compositional Patterns of Non-Functional Properties

This section shares material with the Journal of Software article "Compositional Patterns of Non-Functional Properties for Contract Negotiation". Like the previous section, it is related to a part of Hervé Chang's PhD Thesis and a collaboration contract with France Télécom R&D (now Orange labs).

While the CBSE approach successfully dealt with the functional dimension of components, one of the major challenges was to facilitate the management of *non-functional* properties [BBB+00]. These properties represent various qualities of software components and systems, and with the proliferation of components in long-running applications, where these qualities are important, the need for identifying and handling these properties as precisely as possible during the design, (re-)configuration and runtime phases is crucial.

3.2.1 Objective

In this context, our goal is to provide a fine-grained representation of a large class of non-functional properties in systems built with hierarchical software components, and to use them to precisely specify and manage these properties. Our approach is to provide both a model and a supporting runtime infrastructure, so to stand half-way between analysis techniques and dedicated monitoring systems. We thus propose to reify some non-functional properties in relation with components, and to provide means to support a basic form of compositional reasoning that relate system properties to component properties. This should then enable software architects to better master the design, integration and also the runtime management of non-functional properties into component-based systems.

The proposed representation consists in some architectural patterns that model non-functional properties. They are based on a classification of some low-level observable non-functional properties, which is established by considering their nature and life-cycle. The proposed patterns reify different kinds of parameters on individual components, as well as physical resources, and are mapped to the general *Fractal* component platform [BCL+04, BCL+06]. We particularly show how the compositional support for non-functional properties is used to conduct the propagation of contract negotiation down into the component hierarchy. Some kinds of efforts can then be realized to react to contract violations related to those properties.

3.2.2 Classification of Non-Functional Properties

The proposed classification is not intended to be exhaustive. It is rather limited to the range of low-level properties which are measurable and sufficiently orthogonal to functional aspects. Hence, non-functional aspects which concern high-level properties and system life-cycle at development and maintenance phases, are not taken into account here, as well as other temporal aspects, which require more knowledge about the behavior of components. Moreover, to reason compositionally on non-functional properties, the analysis of non-functional properties of a system must also be based on properties of the components that compose it. Consequently, the classification also takes into account the composability of properties at the level of component compositions. The classification is then built by first analyzing what kinds of non-functional properties can be directly derived from individual components, and what kind of features they express in relation with them. The life-cycle of properties is also analyzed in relation with the one of components. The moments when these properties are defined are distinguished, as well as when they are to be observed.

The proposed patterns directly match the categories of non-functional properties at an abstract level of architecture specification in order to remain independent from underlying component technologies. They are also used to support automated reasoning on compositional properties, once and for all, at the level of patterns themselves.

Some non-functional properties represent the key features or *nature of components*. For example, they can describe a memory footprint, the compatibility version number of a video codec or a maximum capacity. These properties are comparable to the technical characteristics of electronic or mechanical components, they capture and represent some design and development features of components and have an impact on their whole usage. They result of choices made when designing and developing components, and are generally taken into account at assembly and configuration times to evaluate the suitability of components, for instance when selecting potential components and matching them to requirements. As they describe the nature of components, they thus cannot be changed and their measurements remain constant all along configuration and runtime phases.

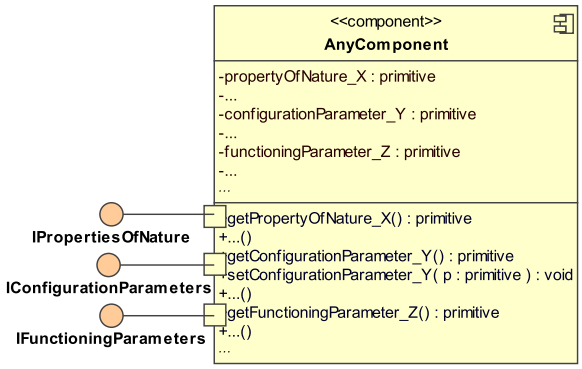
Some other properties represent *configurable parameters* of components themselves. They describe, for example, the size of a buffer component or the maximum size of a resource pool component. These properties influence the required and provided services of components, and may be (re-)parametrized to set components to some specific functioning mode. They can be defined during the development phase by default values, but they are mostly observed and changed at assembly and (re-)configuration times, to properly customize the functioning mode of components, and also adapt them to their runtime environment (other components, runtime infrastructure). Once set, these properties are generally defined to remain constant between two successive reconfiguration phases. Compared to the previous category, configuration parameters are defined to adapt components, they thus support a wider range of change, but still remain constant all along an execution.

Others properties describe *functioning parameters* of components. For example, they can describe the number of active sessions on a web server, the current state of a video player processing a media stream, and the current number of packets exchanged between a given client and a server. These properties capture some key information about the behavior of components at runtime and can thus be seen as properties that probe for some functional aspects of components. As they are related to the runtime behavior of components, they are naturally defined and observed during this time. Compared to the previous property categories, functioning properties pinpoint some elements related to the varying behavior of components at runtime.

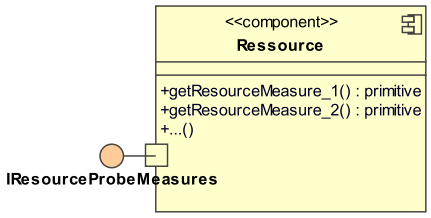
The two other categories considered in our classification consist in properties which are related to the runtime infrastructure. These properties can represent physical resources such as memory, CPU as well as other network properties (bandwidth). They are generally considered under the general term of *resources*, and they describe exogenous requirements of applications. They clearly determine the execution of services, in term of external resources being provided by the underlying infrastructure. In order to make possible their effective monitoring and management, and take advantage of the component-based approach, these resources are now commonly reified at the application level. The resource properties are completely defined at deployment time, as they refer to the deployment infrastructure and runtime environment, and for a given deployment, their existence is constant all along the runtime phase, and until the next deployment. Compared to the three previous property classes, resources concern the underlying infrastructure, and are not directly attached to business components. Beyond the existence of resources, the properties that are frequently considered are *resource capacity*

properties, which describe some aspects in relation with the resource use, as memory occupation, battery remaining capacity, CPU usage, or bandwidth level. These properties express and quantify the level of resources provided by the underlying infrastructure, and required by applications. They are the most often considered resource property, as they represent critical properties of resource-constrained applications, and strongly influence their runtime qualities. As they describe resource consumption at runtime, these properties are defined and observed at this time.

3.2.3 Modeling Patterns



(a) Patterns for properties of nature, configuration parameters and functioning parameters.



(b) Patterns for resources and resource capacity properties.

Figure 3.6: Overview of modeling patterns.

The proposed patterns are described using *UML* 2 component diagrams [RJB04], so that they can be more easily applicable to different component platforms. *UML* 2 components represent independent, interchangeable parts of a system. They realize one or more provided and required interfaces, which determine the behavior of components. Interfaces define sets of operations that components implement. Attributes can also be added to components to represent data fields or properties about them.

The first three categories of non-functional properties directly match the concept of attributes. As properties of nature cannot be changed, they are modeled as read-only private component attributes, which are accessed only by their associated getter operations defined in a provided interface (named `IPropertiesOfNature` in Fig. 3.6a). Configuration parameters can be defined and also re-parameterized. They are thus modeled as both read and write component attributes which are accessed and modified through their associated getter and setter operations defined in a provided interface (named `IConfigurationProperties` in Fig. 3.6a).

Functioning parameters provide information about some aspects of the behavior of components. They are modeled as read-only component attributes with their associated getter operations defined in a provided interface (named `IFunctioningParameters` in Fig. 3.6a). One may note that, at the modeling level, the integration patterns proposed for the three previous property classes are quite similar, as they only relate to the structure of components. However, the semantics of each pattern is rather different, as they each have their own definition on how and when properties are defined and observed (see section 3.2.2).

Physical resources are elements from the underlying infrastructure as a whole. To keep this design both at the infrastructure and application levels, resources are reified as full-fledged components. This then allows one to manipulate these reified resource components as usual business components, and to make them work together seamlessly. In particular, as resource capacity properties probe some information about the level of resources provided by underlying physical resources, they are modeled as operations of the reified resource components, and are accessed through a provided interface (named `IResourceProbeMeasures` in Fig. 3.6b).
As components may exhibit properties that belong to several of these previous categories, the proposed patterns can be applied together. Regarding the measurement of non-functional properties, the proposed patterns provide standardized way to represent properties at the modeling level, but they do not provide predefined mechanisms to measure them, at the implementation level. Such mechanisms are defined at implementation time when the component technology and the runtime environment are determined. For example, runtime properties, mainly related to resources, can be simply measured through appropriate resource probes and asynchronous communications support for processing monitoring information, like the ones provided in the DREAM framework [LQS05].

3.2.4 Reasoning Support for Compositional Properties

To be able to reason on the realization of a compositional property from other ones, information describing the relationships, as well as some appropriate reasoning support must be provided.
Except resource properties, which do not express quantifiable properties, other properties from our classification have been modeled using patterns that derive directly from individual components. These properties are thus compositional by nature, and some simple form of compositional reasoning can be supported. Building on those primitive parts, we define a *compositional property* as a property providing the following characteristics: *(C1)* the set of properties that contribute in decomposing that property, *(C2)* for each contributing property, the components that realize it, and *(C3)* a composition function that allows for computing the overall value of the property given each contributing property. To illustrate this, Fig. 3.7 gives two examples of simple compositional properties. Fig. 3.7a describes a compositional relation A.Interface.p=B.Interface.p that links the assembly property `p` of `A` directly to the same property `p` on its subcomponent `B`. This property is realized by the subcomponent `B`, and these two properties are equal (identity function). In Fig. 3.7b, the compositional relation A.Interface.p=f(B.Interface.p1,C.Interface.p2) now relates the property `p` of `A` to the two properties `p1` and `p2` ascribed to `B` and `C`. The set of contributing properties of `p` are now `p1` and `p2`; they are respectively realized on the subcomponents `B` and `C`, and the composition function is `f` (which can be, for example, a simple arithmetical sum or min function).
For a compositional property, determining these *compositional information* formally may be hard, even impossible, and may also require deeper analysis, especially as the composition function (*C3*) may be difficult to express. The fact that our study is restricted both to non-functional properties that are directly derived from single components, and compositional properties, which are a function of properties of its components only (no system environment, or architecture-related factors), simplifies identifying these information. Even with these hypothesis, there are still many dependencies between properties as well as measurement influences during monitoring (a form of the observer effect, at least because monitoring consumes time and space), which are hard to express and model using simple compositional functions. Compositional functions are to be defined individually for each considered compositional property, and it is necessary to provide a trade-off between simple and provable formula and complicated but more error-prone functions. In our case the composition functions can be defined with almost arbitrary code, as these functions are specified with assertion-based contracts

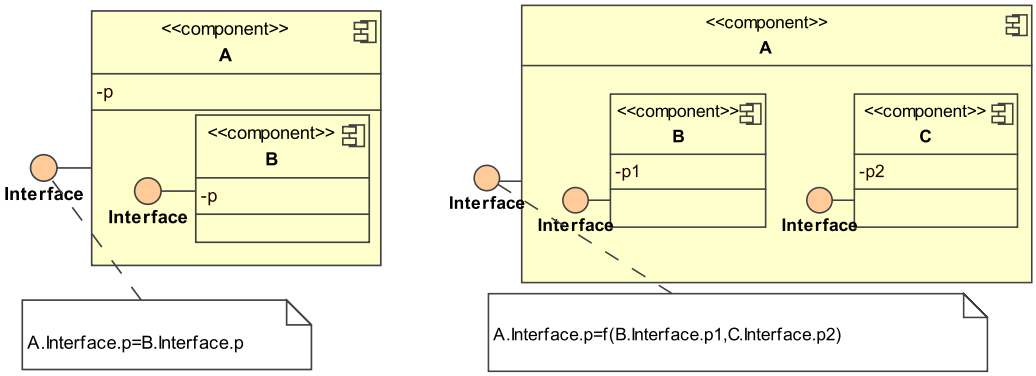


Figure 3.7: Examples of compositional relations.

(see section 2.1), so that checking can be performed during both testing and exploitation stages. We thus suppose that for each compositional property, compositional information are provided through descriptive meta-data that may be expressed using code annotations or Domain Specific Language (DSL) facilities.

To enable reasoning on them at runtime, compositional properties are reified as meta-objects, and we integrate the following compositional properties support. A *CompositionalPropertyManager* (see Fig. 3.8) is built for each *Component* to manage the set of its compositional properties. It uses the provided compositional information to instantiate and register a *CompositionalProperty* (meta-object) for each compositional property. Each property object reifies a corresponding compositional property and it gives access to all of its compositional information: its value, all other properties necessary to compute its composition function and their contributing components.

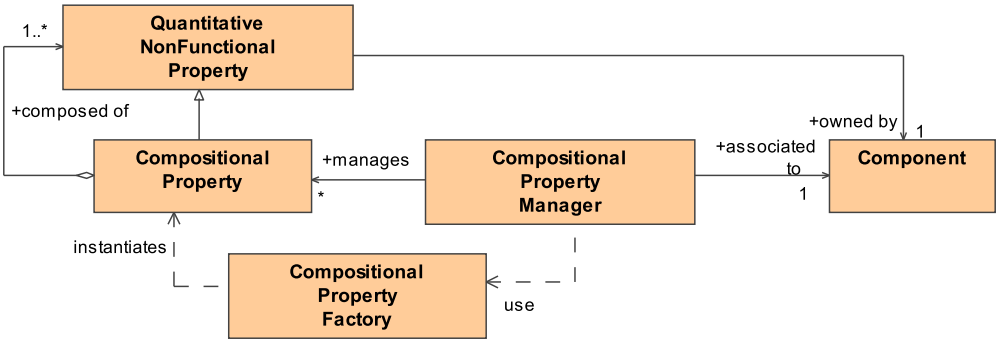


Figure 3.8: Elements at the meta-level.

At runtime, the *CompositionalPropertyManager* can be exploited by elements at the base or at the meta level to retrieve the corresponding *CompositionalProperty* object and get the compositional information about the property it reifies. When interacting with the compositional manager, the sequence of messages carried out by a client is as follows: i) the client invokes the compositional manager that manages the property objects of its associated component to look up the property object

that corresponds to a given compositional property, ii) the compositional manager returns the reference to the property object, iii) the client can then directly invokes the property object to retrieve, for example, the value of the property computed from the compositional function or the list of the contributing components that decompose the property. Other compositional information are retrieved similarly.

3.2.5 Mapping Patterns to the Fractal Platform

We now illustrate how the proposed patterns can be integrated and used into a component platform such as the *Fractal* component model.
The *Fractal* component model provides dedicated *attribute-controller* interfaces to model orthogonal properties of components. They give access to component attributes before starting components and without needing to bind and use their functional interfaces. They also offer various access modes (read and/or write accesses) which make it possible to respect the difference between properties of nature and functioning properties, which cannot be changed, and configuration parameters which can be modified. Hence, as *properties of nature*, *configuration parameters* and *functioning parameters* of components, are simply modeled through component attributes, they are basically mapped to *Fractal* attributes and attributes control interfaces (see Fig. 3.9a and 3.9b), with their appropriate read-only or read-and-write operations.

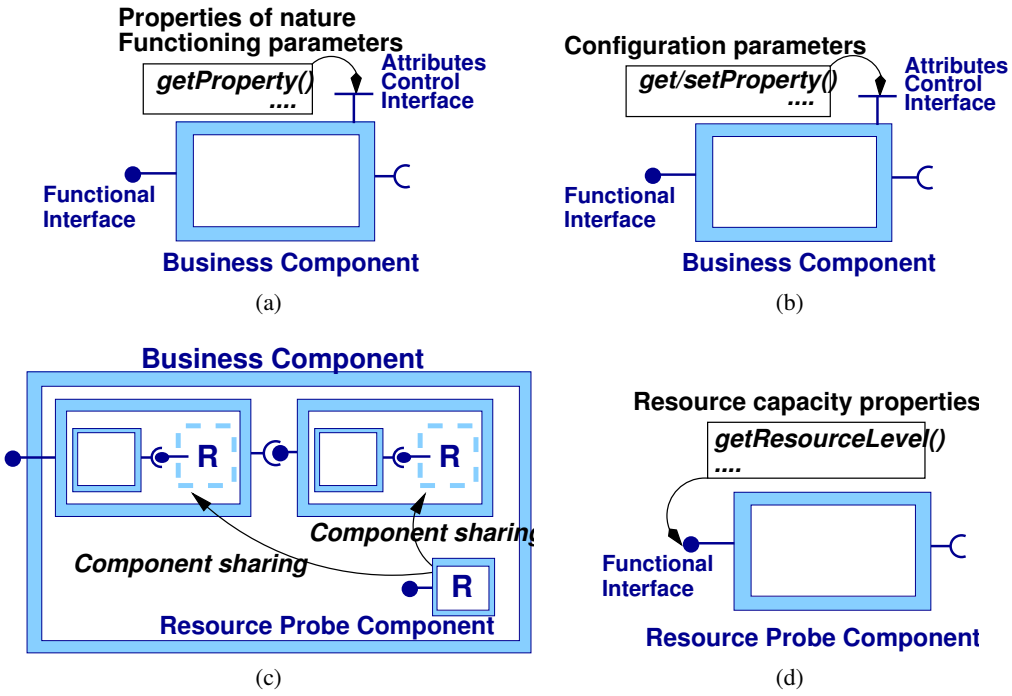


Figure 3.9: Patterns for Fractal components.

As *resources* are modeled by components, they are reified using full-fledged *Fractal* components, which do not provide *a priori* advanced services, except for probing their associated physical resource. Moreover, as resources can be used by several distinct business components, the *component sharing* feature provided by the *Fractal* platform is exploited to reflect resource sharing at the com-

ponent composition level (see Fig. 3.9c). It enables one to use a same instance of a reified resource component, in several distinct enclosing components, at different level of hierarchy, while preserving component encapsulation.
Resource capacity properties represent primitive data collected by resources probes, which are modeled using *Fractal* components. They are modeled using functional interfaces attached to the corresponding resource probe component (see Fig. 3.9d). The type of collected data is open and to be determined by the developers of probe components, spanning from primitive measurements to more advanced performance indicators processed by statistical models (interpolations, correlations, etc.). It should be noted that, as *Fractal* components can be recursively nested, the proposed integration patterns can be applied at any level of hierarchy in a uniform way.

3.2.6 Illustration

To illustrate our contribution, we use a variant of the Amui system already presented in section 2.2.2. The Amui system manages automatic grouping of users, according to their common interests, and dynamic application sharing. Several variants of the system were developed to study both *Fractal* capabilities and the different propositions we designed and prototyped. The version we use here automatically groups users into chat rooms, and streams videos to grouped users according to their common interests.
The server, shown on figure 3.10, is represented by the composite component `FractalInstant-Communication`, which is formed out of three subcomponents : `InstantGroup` manages the users and their grouping through its provided interface `UserMgmt`[20], `VideoService` manages the video streaming service, and `BdwMonitor` monitors the network bandwidth and measures the overall bandwidth consumption of the server. `InstantGroup` is composed of `UserManager` which manages the users, `GroupManager` which manages groups, `MsgMonitor` which monitors messages exchanged between users, and `InstantFacade` which pilots the other components. `InstantGroup` also exhibits the properties `maxUsers` and `groupedUsersRatio` which, respectively, describe the maximum number of concurrent users that the server supports, and the rate of users that have been grouped. `VideoService` is composed of `VideoManager` which manages the video streaming, and `VideoMonitor` which monitors the bandwidth consumption of the video service. Moreover, each component is endowed with capabilities to control its bindings, content and lifecycle (respectively depicted as `BC`, `CC` and `LC` in Fig. 3.10). The content of `BdwMonitor` is detailed together with contracts in section 3.2.7.
In this example, the various proposed categories are illustrated. The `maxUsers` and `grouped-UsersRatio` properties of `InstantGroup` are respectively a configuration parameter that is defined when configuring the server to set the maximum threshold of concurrent users, and a functioning parameter that describes the rate of users that have been grouped. They are modeled with read-and-write *Fractal* attributes. The property `nbUsers` of `UserManager` is a functioning parameter that describes how many users have been registered. It is modeled with a read-only *Fractal* attribute. The property `nbGroupedUsers` of `GroupManager` is a functioning parameter that describes how many users have been grouped, and is also modeled with a read-only attribute. All these attributes are accessed through the attributes controller interface of their corresponding component. As for the network bandwidth property, the associated resource probe is modeled as a *Fractal* component (`BdwMonitor`), and the probed data, such as the level of network bandwidth used (`getBdWidthLevel()`), are modeled through the functional interface `BdWidthInfo`. More-

[20]For lisibility sake, this interface is not detailed on Fig. 3.10.

over, as `BdwMonitor` relies on the level of bandwidth used for the messaging and video service to compute the overall bandwidth, it uses `MsgMonitor` and `VideoMonitor` which are then shared and hosted in `BdwMonitor`. For lisibility sake, the sharing of these components is further detailed in section 3.2.7.

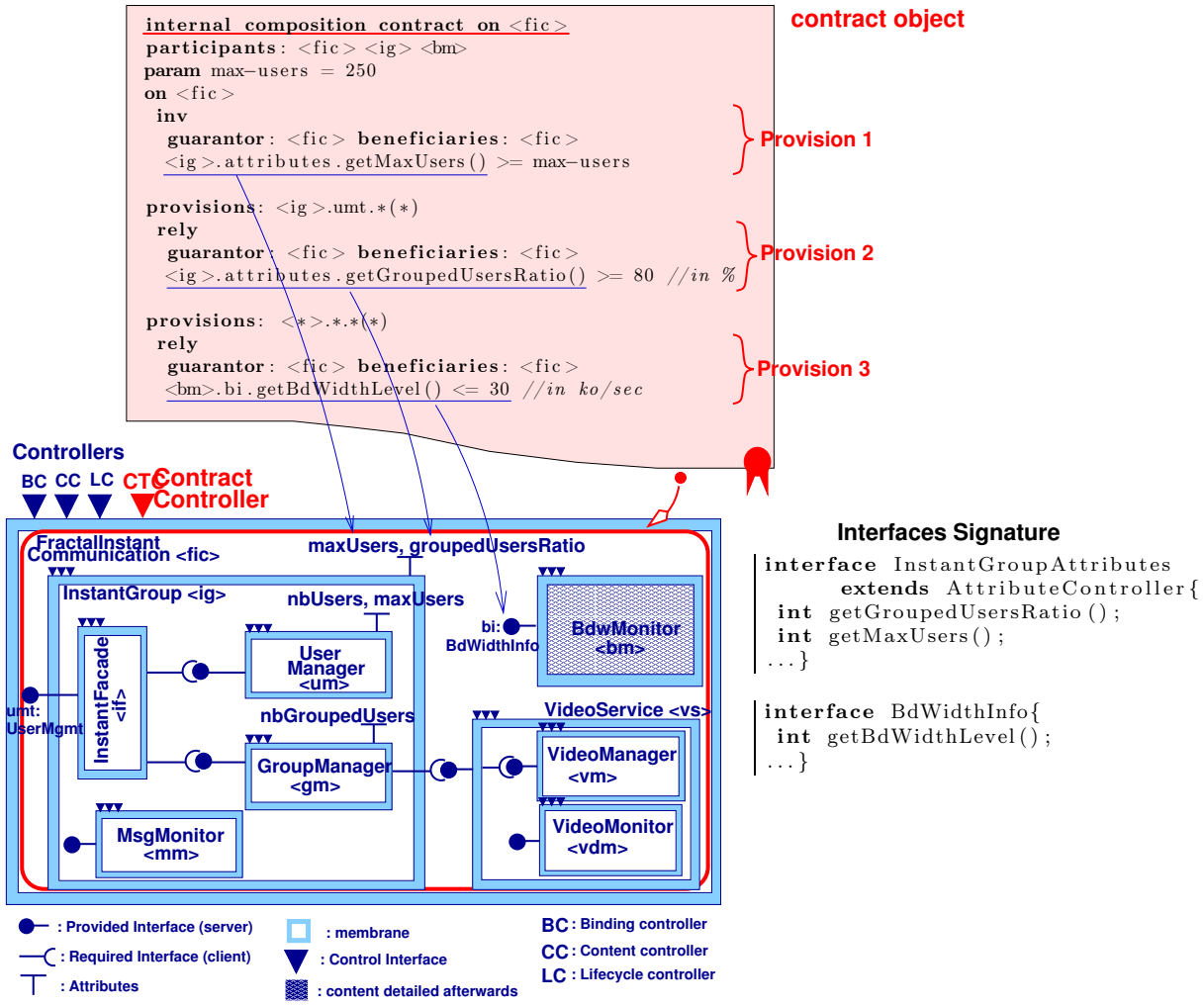


Figure 3.10: Architecture of the server, and some contracts.

3.2.7 Exploitation in Contract Negotiation

We now illustrate how the effort-based negotiation policy defined in section 3.1.5 can rely on the proposed integration patterns and the compositional properties support. This negotiation policy consists in exploiting the responsibility of the *guarantor* component. As it is responsible either of its assembly or the implementation of some terms referred in the negotiated provision, the guarantor can act to restore the validity of the provision by either doing some reconfiguration actions at its level or propagating the negotiation process down its hierarchy. In this latter case, *some contributing components*, which contributes to the negotiated provision at the lower levels, are then consulted to propose some *efforts* that may revalidate the violated provision at the higher level. The contributing

properties that decompose a given compositional property are used to propagate the negotiation from a level of component hierarchy to the sub-level. They are identified using the compositional function. Moreover, for each contributing property, its realizing component is explicitly identified according to the proposed integration patterns. Each contributing component which is asked for efforts, may then propose some changes regarding the property it realizes.

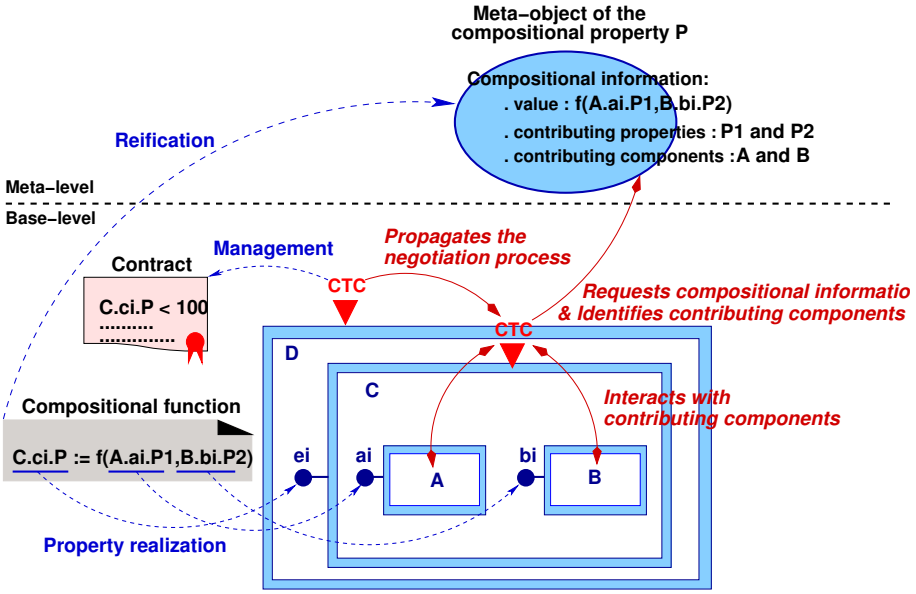


Figure 3.11: Overview of the propagative negotiation process.

Fig. 3.11 summarizes the propagative process by depicting the various entities involved and their interactions during the compositional reasoning. The negotiation focuses on a contract provision built from the specification C.ci.P $<$ 100, which expresses a maximum threshold for A property P realized through an interface ci of a component C. The contract is managed by the contract controller (CTC) of the component D, which then activates an atomic negotiation in case of violation. Following the proposed compositional support (see section 3.2.4), the meta-object of the compositional property P is built from the compositional function C.ci.P := f(A.ai.P1,B.bi.P2) and it gives access to the compositional information. It is exploited by the contract controller of the component C to retrieve the set of contributing components, A and B, in order to consult them, and requests efforts from them according to their contribution in the contract provision.
Moreover, when the checking of a contract provision fails, the overall negotiation process using the propagative negotiation policy involves the contract controller, which manages the violated contract and the guarantor component. It then executes according to the following steps:

1. The contract controller requests proposals from the guarantor component. In response to these requests, the guarantor component can then either make proposals to revalidate the provision at its level or, decide to consult some components in its content (if it is composite) ;

2. In this latter case, the contract controller of the guarantor takes in charge the negotiation and thus have to identify the set of components that contribute in the property that is specified in the contract provision, and that are to be consulted ;

3. These components either implement the property, or belong to the set of components that contribute in decomposing that property. In the first case, this contract controller uses the integration patterns to identify the component that implement the property. In the other case, it interacts with the compositional properties manager and the associated compositional property meta-object to retrieve the compositional information that describe the decomposition of that property;

4. Once identified, these contributing components are consulted by the contract controller in order to make proposals that may revalidate the violated contract provision. At their turn, they can either propose changes that may revalidate the contract provision, or take in charge the negotiation and propagate it in their content, following the process as in step 2.

Some contracts on non-functional properties

Back to our working example, Figure 3.10 shows a pretty print of the reified *internal composition contract* which is built in the content of the component <fic>. This contract is managed by the contract controller (CTC in figure 3.10) of <fic>, and contains three contract provisions which express some internal behavior rules on the implementation of <fic>.
The first provision (see figure 3.10) defines an invariant on the configuration of <ig>, such that the maximum threshold of concurrent users that the server can support (maxUsers) is higher than 250 users. The second one constraints <ig> by defining a minimum threshold of 80% for the groupedUsersRatio (on 10 registered users, at least 8 of them must belong to a group), which must hold all along the execution of every method of <umt> (rely construction and operator $*$). The third provision constraints <bm> by defining a bandwidth consumption threshold of 30ko/sec, which is required to prevent a high bandwidth use. This constraint must hold all along the execution of every method in the content of <fic>. As for checking, the first contract provision is checked at configuration time, as it specifies an invariant of the configuration. The other provisions, which specify functioning and resource capacity properties, are checked at runtime. Regarding responsibilities, for each of these three contract provisions, <fic> is at the same time, the guarantor and the beneficiary component, as it has to take in charge its internal assembly and also benefits from it (cf. section 2.1.5).

Scenario for a configuration parameter

The first provision may be violated, if for example, the component <ig> supports by default a maximum threshold of 100 users. Let us suppose that the compositional relation ($R1$) (see figure 3.12) is provided to describe the fact that the property maxUsers of <ig> decomposes itself identically into the same property maxUsers of <um>. The negotiation process then involves the contract controller (CTC) of <fic> and <fic> itself, as the unique guarantor. It executes as follows. First, the contract controller consults the component <fic> and requests from it some proposals (*step 1*). As <fic> is responsible of its internal assembly, it then takes in charge the negotiation process and consults its subcomponent <ig>, which carries the property maxUsers (*step 2*). To propagate the negotiation in its content, the contract controller of <ig> interacts with the compositional properties controller (named CPC in figure 3.12) and the compositional property meta-object associated to the maxUsers property, to identify the components to be consulted. The compositional information that describe the maxUsers property (*step 3*) are then retrieved, and the component <um> is identified as the unique

contributing component. The contract controller of <ig> requests proposals from <um> which, according to its negotiation reasoning, makes proposals that may, for example, consist in reconfiguring its parameter with a higher value of maxUsers (maxUsers=300 for example) (*step 4*). For each proposal, the contract controller of <ig> uses the compositional function to evaluate whether the proposed changes are sufficient to revalidate the contract provision.

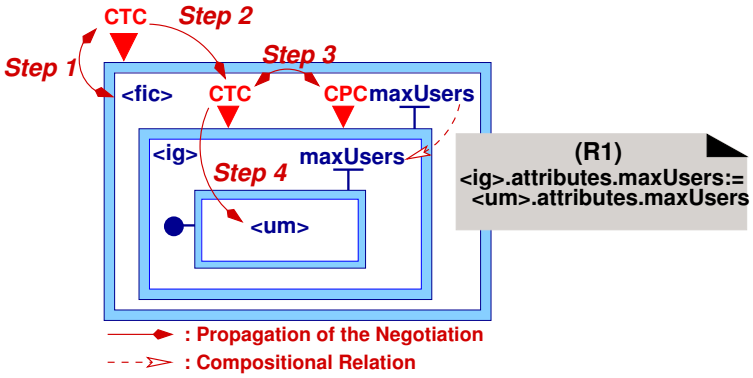


Figure 3.12: Propagative scheme for the maxUsers property.

Scenario for a functioning parameter

The second provision may be violated if the grouped users ratio is lower than 80%. To negotiate this, let us suppose that the compositional relation ($R2$) (see figure 3.13), describes the decomposition of the property groupedUsersRatio of <ig> into the property nbGroupedUsers of <gm> and nbUsers of <um>. It expresses the fact that the grouped users ratio is equal to the ratio between the number of users in groups and the overall number of users. The negotiation process involves the same negotiating parties as in the previous example, and it is propagated at the level of <ig>, using the same propagation scheme (*step 1* and *2*). However, the components that contribute here in realizing the property groupedUsersRatio are <gm> and <um>, which respectively exhibit the property nbGroupedUsers and nbUsers (*step 3*). They are thus consulted (*step 4*), but, as these properties describe functioning properties of <gm> and <um>, they cannot be directly changed. <gm> and <um> are likely to be unable to propose some efforts. The negotiation then terminates with a failure, which leads to an exception. This exception may be caught outside any negotiation process in order to perform more *adhoc* and global adaptations or reconfigurations of components (replacing the <gm> component, etc.).

Scenario for a resource capacity property

The third provision is challenged if the global bandwidth consumption exceeds 30ko/sec. As the BdwMonitor component relies on the bandwidth levels of the messaging and video services, which are measured by the probe components <mm> and <vdm>, these two components are shared and their associated slave instances, <mm'> and <vdm'>, are hosted in the content of BdwMonitor. Besides, the compositional relation named ($R3$) in figure 3.14 is provided to describe that the property bdWidthLevel of <bm> decomposes into the sum of the property bdWidthLevel of <mm'> and <vdm'>. As in the two previous scenarios, following the steps 1 and 2, the contract controller of <bm> takes in charge the negotiation and identifies the component <mm'> and <vdm'> as the

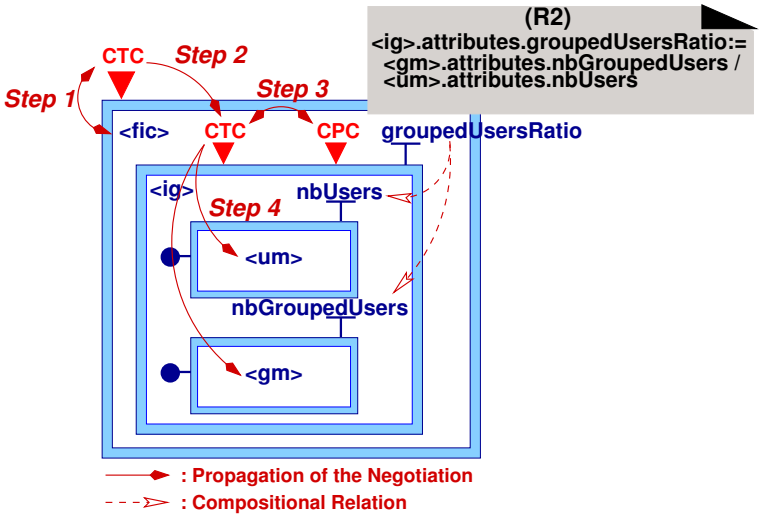


Figure 3.13: Propagative scheme for the `groupedUsersRatio` property.

contributing components. However, as opposed to previous examples, <mm′> and <vdm′> are not consulted as they merely represent the slave instances of resource probe components, which cannot propose efforts at the application level. The sharing relation is then exploited to retrieve the reference to the master probe components <mm> and <vdm> (*step 4*), from which the enclosing business components <ig> and <vs> are retrieved (*step 5*). The components <vg> and <ig> are then consulted by the contract controller of <bm> (*step 6*), as they are the components at the application-level that may propose efforts to lower the bandwidth consumption (selecting lower bitrates, changing used codecs, compressing file transfers, etc.).

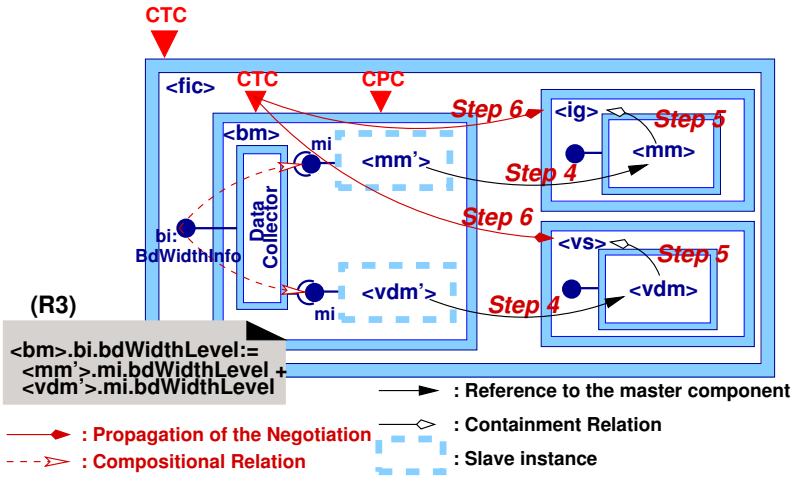


Figure 3.14: Propagative scheme for the `bdWidthLevel` property.

3.2.8 Implementation

Like the *ConFract* prototype and the negotiation mechanisms associated to it, the compositional properties support has been implemented with the `Julia` version of *Fractal*, which notably supports

component membranes, i.e. controllers implemented as components themselves.

Support of compositional properties. As the control dimension of *Fractal* components is open, the *CompositionalPropertyManager* is implemented as a new *Fractal* controller component, namely *compositional property controller*, and integrated in the membrane of every composite component to control its set of compositional properties.
For each compositional property of a component, the meta-data representing the compositional information are described using Java annotations, and a custom multi-value annotation type is defined to describe: the set of contributing components, the set of contributing properties, the modeling elements which realize each property (interface and method names) and the compositional function that links the value of the compositional property to its contributing properties. These annotations are then parsed and the compositional information of each property are retrieved by the *compositional property controller* in order to instantiate and register each corresponding property object. This controller thus maintains the references to all of the compositional property objects of its component, and like any other *Fractal* controllers, its control interface is accessed, at runtime by introspecting the given component, and used to access to each compositional property object.

Negotiation mechanisms and propagation. The negotiation mechanisms are organized around atomic negotiations. In our *Fractal* implementation, a dedicated composite component is dynamically built to represent the atomic negotiation process (cf. section 3.1.7). This component is hosted in the membrane of the component whose contract controller initiates the negotiation process (see figure 3.15). It encapsulates some proxy components to the various parties that have to negotiate. when a negotiation needs a propagation, it creates a new atomic negotiation. As the spatial scope

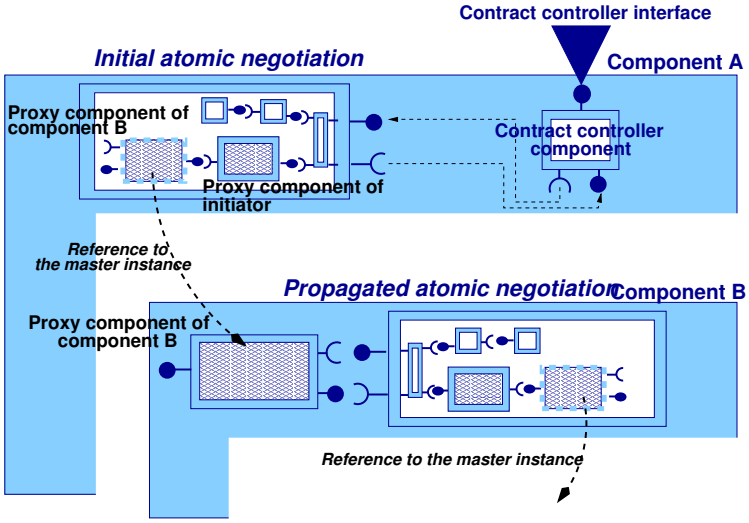


Figure 3.15: High-level architecture of atomic negotiations in component membranes.

of an atomic negotiation involves only parties that are related by at most one level of hierarchy of component, a new atomic negotiation component is then built for each propagated negotiation. The construction of this new component is activated by the participating component which have proposed the propagation. It thus becomes client of the new atomic negotiation component, and can both re-

ceive the negotiation outcome of the propagated atomic negotiation component and make proposals in the negotiation at the upper level (see figure 3.15).

3.2.9 Related Work

Numerous studies have been conducted around the analysis, modeling and management of non-functional properties. At the highest-level of analysis, some approaches provide methodologies to analyze quality attributes [BKLW95], or address non-functional properties through quality standards (IEEE 1061, ISO/IEC-9126) and models [BV02, BRO+02] that provide classifications of high-level properties. They aim at proposing a generic taxonomy and studying relationships between properties, or structuring knowledge by successively defining and decomposing non-functional characteristics. At the lowest-level of platform and resource management, substantial works exist on providing resource management and monitoring capabilities to applications, at different level of abstractions (API, technologies) [CHS+05]. Such works also aim at integrating advanced tools for the diagnosis of performance issues [PS06]. Compared to these works, our work stands in between analysis techniques and management systems. We focus on providing some patterns to finely model and integrate a range of non-functional properties in software components, so that it is possible to precisely manage them at runtime, and also reason on their composition according to those of components.
To achieve non-functional requirements in the domain of distributed systems, numerous works proposed some component-based middleware platforms that provide QoS control and measurement capabilities through reflective and adaptive techniques [BAB+00, N. 01]. However, they particularly focus on critical network-related properties, and provide integrated control mechanisms without explicit representation of non-functional properties. Some other component platforms also enable flexible integration of arbitrary non-functional services using code transformation such as aspect-weaving, or indirection frameworks (interceptors, meta-object protocols). Non-functional services are essentially middleware-related services (transactions, load balancing, security checks, etc.), and they are handled by containers which wrap set of components.
Specific to component-based systems, several compositional approaches aim at improving non functional property analysis in component assemblies. Analysis models and property theories are thus integrated to component technology [HMSW03], and they allow one to guarantee, by construction, the predictability of some properties on component assemblies. However, they require advanced analysis models and techniques, and are mostly dedicated to specific properties, such as latency [HMSW03], reliability [HMW01, RSP03a] or memory usage [EFHC02]. Some of these models could also be extended to other properties if the properties are properly related to the architecture and modeled in some generic ways that make possible to reason on them. Our approach differs as, instead of analyzing formally some specific property upon existing theories, we rather focus on a larger range of low-level non-functional properties which can be directly modeled and integrated to runtime components and platforms so that mechanisms to manage and monitor these properties can be developed.
To enable reasoning on non-functional properties at the architectural level, the relationship between software components and software architecture has been outlined [WSHK01] and exploited by studying how properties relate to component assemblies and individual component properties. In particular, in order to help describing how properties relate to compositions, an interesting classification [CLP05] has synthesized different classes of dependency between properties, components and their context. Our approach aligns with these works, as it integrates non-functional properties categories at the architecture level using existing basic elements of components (components, component attributes, interfaces). However, by focusing on some low-level properties only related to individual component, we only support some simple forms of compositional properties which are a function of

the properties of the components involved (no architecture, context or usage dependencies).

3.2.10 Summary

We defined abstract integration patterns that map low-level measurable non-functional properties to individual software components. Properties are then distinguished among attributes of nature, configurable parameters, functioning parameters, resources and resource capacities. These patterns are then mapped to the *Fractal* hierarchical component platform. As properties are clearly modeled to components and directly derived only from them, some simple compositional relations, which describe the realization of properties given component compositions, can be expressed. Elements at the component meta-level have also been provided to support reasoning on such compositional properties. We also showed how these patterns and the compositional support are exploited to negotiate non-functional contracts on *Fractal* hierarchical components. They are used in a general propagative scheme, which, by following the compositional relationship between properties, propagate the negotiation to contributing components, so that they may propose efforts to revalidate violated contracts. Both the patterns and the negotiation support have been developed and validated on different variants of the Amui system, already described in the previous section.

3.3 Self-adaptive QoI-aware Monitoring

This section shares material with the ADAPTIVE'10 paper "A QoI-aware Framework for Adaptive Monitoring" [LDCMR10]. It concerns Bao Le Duc's PhD Thesis and collaborative work with Jacques Malenfant and Nicolas Rivierre within a contract with Orange labs.
In chapter 2, we have presented contract-based techniques and tools aimed at improving reliability of service and component based architectures. This follows the overall objective of taming complexity of the new forms of software intensive systems, which are very large, highly distributed and 24/7 operated. Besides these systems are now more and more organized around *Service Level Agreements* (SLA), a form of contract which can be relevant both at the business and infrastructure levels. They mostly refer to *Quality of Service* (QoS) dimensions to express guarantees and thus call for underlying monitoring systems.
But these systems are not the only ones that need monitoring subsystems. With activities such as scheduling, resource allocation and problem diagnosis, there are similar needs for continuous monitoring of all parts of software intensive systems, from the underlying infrastructures to high level services.

3.3.1 Objective

With different clients needing to monitor a large number of QoS dimensions, the issue of the *Quality of Information* (QoI) arises. QoI is the expression of the properties required from the monitored QoS data [BKS03]. It can be about the type of the monitored data, their granularity, lifespan or simply their precision. With highly distributed and pervasive systems, monitoring is now always done over a distributed infrastructure. It must extract information among deployed processes, efficiently collect and redistribute them to the querying clients adapting formats when needed, while dealing with communication delays, non deterministic event ordering and usually an alteration of the observed system [JLSU87].
In SOA, monitoring systems have been quickly provided [BGP06, BTPT06], but with implicit QoI support [MRD08] or no support at all. In this work, our viewpoint was to consider that a monitoring

system should currently provide several information flows to multiple clients having specific QoI requests, while being dynamically reconfigurable. We also considered that this capability of dynamic reconfiguration should be applicable to the monitoring system itself, constraining it on the resources it consumes. Several works have been conducted in different domains such as context-aware computing, data stream processing or transactional systems [ACC09, JYDZ09, MW06], with focus on QoI or adaptive monitoring, but not taking into account all our requirements at the same time.
We thus developed a monitoring framework, ADAMO (ADAptive MOnitoring), which is able to deal with multiple clients requesting flexible and dynamically reconfigurable access to dynamic data sources with different QoI needs, and supports automatic configuration of all monitoring entities and data sources so that QoI and resource constraints are taken into account.
To illustrate the requirements of our monitoring framework, we take here a small example of two clients requesting data with different QoI characteristics. In [LD10], we used scenarios from a C^3 (Control, Command and Communication) application, i.e. a system that mediates commanders and their teams on an action field (rescue, battle, etc.). There are thus needs to gather data on positions of teams, transports and materials, and on various field sensors, with low bandwidth consumption on wireless networks. The two clients can have the need for different data, some of them being common, with quality such as coherency, i.e. values taken in some time interval, or freshness, i.e. the age of data does not pass some limit. For all clients, data access must be efficient and flexible enough, masking any specificity that can be automatically handled such as data conversion. Finally, regarding the consumption of the monitoring system itself, we consider it can be collocated in the same infrastructure and that its bandwidth consumption can be constrained, e.g. not to take more that 5% of the total bandwidth. Consequently, the monitoring system must also be aware of bandwidth fluctuation to adapt itself.

3.3.2 ADAMO Underlying Model

ADAMO relies on a QoI model that formalizes *data sources*, *monitoring queries* and *system resources* and which was mainly established by Jacques Malenfant. The model leverages constraint solving to find appropriate frequencies to configure data sources according to clients needs and resource constraints. Consumers send ADAMO QoI-aware monitoring queries and receive data streams as result. ADAMO addresses QoI by processing queries so that the requested QoI and resource constraints are automatically transformed into appropriate configurations of the data sources. The formalization of this QoI model is available in [LDCMR10] and detailed in Le Duc's PhD Thesis [LD10]. We only discuss here the main elements of the model and the resulting resolution capabilities.
In the model, monitored values are defined as independent data sources, even though some may actually represent the same physical entity or sensor. A data stream is sequences of data produced in temporal order by a probe, so that each data value is associated with a time-stamp. It can be used to enforce QoI constraints on the configuration parameters, e.g., interrogation mode (push/pull) or sampling frequency. The sampling frequencies act as filters on raw data streams to pick the values that will be transmitted to clients. Monitoring can then be regulated by choosing one of the possible frequencies.
A query specifies the need of a client to receive tuples of data under some given QoI constraints. In the developed version, the ADAMO model addresses the age and coherency properties, but the framework architecture presented below is aimed at integrate easily new QoI properties. Currently, ADAMO addresses two different QoI properties: age and coherency. An age constraint then imposes a maximal delay between the production of a data by a source and its reception by the client. A coherency constraint imposes a maximal delay between any pair of data of the requested tuple.

System resources represents bandwidth, CPU, and memory resources. Each of the resources uses available data source properties expressing the consumption of that resource when delivering data to consumers to get the overall estimation of their consumption by the monitoring system in a given configuration of the data sources.
The objective of the model is to enable reasoning on data source configuration, so that a given set of queries under some resource constraints can be satisfied. Even if the QoI model is generic and open to be extended with new data sources, properties, resource and constraints, each kind of QoI will require a specific processing, which integration would be facilitated by the architecture framework presented in section 3.3.3.
The model resolution is possible in both resource unconstrained and resource constrained cases. If resources are unconstrained, the system is supposed to be able to process all data queries, and the result should be a configuration of each data source that satisfies highest QoI requirements among the set of queries. In the constrained case, the configuration should be a trade-off between QoI requirements and resource constraints. This trade-off problem varies upon usage contexts as well as how QoI impacts on clients. To got beyond simple approaches that would equally reduce QoI for all client, some utility functions can express that some requirements have higher utility than others [PSGS04, ACMS06], leading the monitoring to guarantee higher QoI for certain clients at the expense of reducing it for the rest. In both cases, the frequency of each source is computed to achieve the QoI required by the set of queries.

QoI enforcement in a resource unconstrained system. When resource is not a concern for monitoring, the problem is to find an assignment for all the properties of each data source such that the constraints are satisfied for all sources and queries. The problem can be modeled as a *constraint satisfaction problem* (CSP). CSP is particularly well-adapted to ADAMO, as it provides a systematic approach to the problem, and can be extended with resource constraints or cross-constraints between criterai or new QoI needs. The variables in the CSP are the data source and the QoI properties appearing in the data source and QoI constraints. Constraints on data sources impose restrictions on the domain of the configuration variables of the data source and can be directly used in the CSP. Constraints on the QoI need to be related to the configuration properties of data sources in order to enforce some values for their configuration. Corresponding constraints are added to the CSP.
The obtained CSP is not detailed here but it can have several solutions, as multiple frequencies for data source may match the desired age and coherency constraints of the queries. In this case, the smallest frequencies in the sets of values satisfying all of the constraints are chosen.

QoI enforcement in a resource constrained system. In the constrained case, we consider that a specific amount of resource already modeled is allocated to the monitoring. A straightforward approach consists in adding a resource constraint over the sum of all consumed resources of all properties with this specific amount. The problem is then to find a configuration that satisfies not only the age and coherency constraints, but also this resource constraint. However, the system is now differently constrained, changing the nature of the problem. As the resource constraint may impair the satisfaction of the age and coherency constraints of some queries, the user should be able to express preferences among its queries so to concentrate the resource on the most important queries and lower, if necessary, the requirements of the less important ones.
In order to allow the user to express preferences over QoI properties, the model is enriched with a set of utility functions that extends query definitions. Each monitoring property as an utility function and utilities are combined to get the total utility of a configuration. Age and coherency constraints

can then be seen as minimal requirements, and the problem becomes to find a configuration that maximizes the above utility under the age, coherency and resource constraints.

3.3.3 ADAMO Architecture

Relying on the model described above, the ADAMO architecture consists in several abstractions to allow for building monitoring systems that enforce QoI requirements. These systems are aimed at being deployed on given points of a distributed infrastructure (this also means ADAMO systems themselves are not distributed).
The common operations of all ADAMO instances are basically i) to gather data from sources, potentially distributed, ii) to store them in a buffer system, iii) to process them so that supported properties are enforced on requested QoI while maintaining the threshold on resource consumption, iv) to deliver process data to clients. The ADAMO architecture thus factors out the necessary common structure and behavior from the monitoring specific parts. The framework itself uses different techniques, hierarchical components to represent functional building blocks, interfaces for decoupling all elements either inside the common parts or as extension points and some design patterns to ease integration of specific parts in the framework.

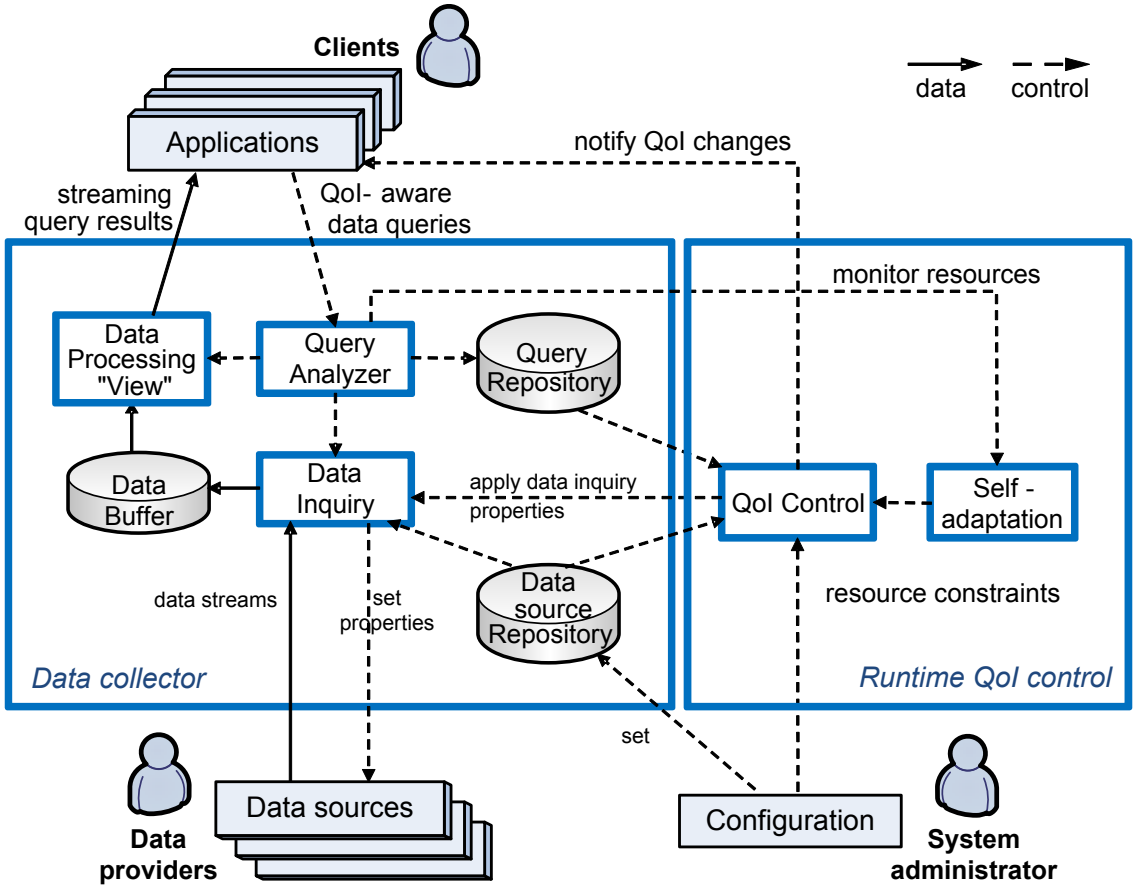


Figure 3.16: ADAMO functional architecture and roles.

Figure 3.16 gives an overview of the main components with data stored and interactions, as well

as roles of people using the framework. There are three main roles in the ADAMO framework. *Applications* are clients of the monitoring system. They request, sometimes dynamically, different QoS data with specific QoI constraints on each of them. Suppliers of the monitoring system are *data sources*, which are also varied in form and location and thus impact differently the bandwidth consumption. Finally, system *administrators* configure the resource constraints over the monitoring system.
Regarding the ADAMO functional architecture, functionalities related to *data collector* are separated from those of (runtime) *QoI control*. In the data collecting part, the client front-end is the *query analyzer* component, which processes QoI-aware data queries of different kinds, submitted as batch or on-demand. Handling queries, the *query analyzer* initiates the *data inquiry* component, which basically wraps an inquiry algorithm that can optimize the connection to data sources by finding an appropriate configure set for their properties (frequency, message size, transmission mode, etc.). The inquired data stream is cached in local *data buffers*. The *query analyzer* also identifies QoI constraints from clients and stores them into a *query repository* for further reasoning. The *QoI control* component then dynamically finds an appropriate configuration for all data inquiries, which satisfies best QoI requirements of concurrent clients given resources constraints. As for the *Self-adaptation* component, it monitors resource levels through the *data collector* part and, if needed, triggers *QoI control* to find a new configuration (see section 3.3.7). Finally, *data processing view*, simply named *view* afterwards, are components that streams data to clients while realizing some final QoI filtering or data conversion.

3.3.4 Elements of the Framework

In ADAMO various abstraction points are available to clarify domain intents and reduce implementation efforts. This allows software architects to focus on solving a problem without being concerned about less relevant lower level details. In the framework, each component represents a level of abstraction that can be extended to specific adaptive monitoring requirements. For example, *QoI control* can be extended in order to adopt a new trade-off algorithm taking into account coherency and some resource constraints.

Query Analyzer. This component is in charge of handling and processing QoI-aware data queries. In ADAMO, a *QoI-aware data query* consists of two specific parts in which (a) a list of dimensions expresses which data sources are to be monitored and how the monitored data are processed, (b) QoI constraints express how good the monitored data should be for the applications. ADAMO then supports two ways to submit a query: static and incremental. In the static mode all queries are submitted to the monitoring service once and for all. A set of data sources (S_Q) is then derived from the set of queries (Q). The incremental mode is obviously more complex and consists in handling queries subscription and removal at runtime. This requires some specific support on existing queries so that data inquiry processes are correctly deactivated. A new set of data sources is then derived from the pre-existent ones and the new query ($S_Q = f(S'_Q, q)$). In both cases, when multiple clients refer to the same data source, the *query analyzer* makes the necessary adjustments to converge to a single data inquiry, so that duplicated remote data transmissions from data sources are avoided. Due to the necessary knowledge on data queries for both the query analyzer and the QoI control component, information related to queries, data sources and their relationships is indexed and stored in a *query repository*.

Data Inquiry. The *data inquiry* component establishes a data inquiry protocol, based on a given configuration C_{S_Q} assigned to data source properties. Data source properties include frequency, message size, data transmission mode (push/pull), but also inquiry mode (batching multiple samples, summary techniques). In practice, message size and data transmission are usually chosen at design time while inquiry frequency is used to regulate data transmission. The monitored data are stored into local data buffers. New updates enrich query results but consume bandwidth as well as computation resources. Therefore, a data inquiry can be adjusted to receive less or more data at runtime in order to achieve QoI and resource consumption objectives.

Data Processing View. A *data processing view* produces high-level abstract information from some low-level raw data. It also provides data to the client applications according to the protocol of their choice (pull or push mode). In most cases, raw data sensed from the environment may be meaningless for clients or some measurements may not be *good* enough for a given QoI request. Two types of data processors are provided to handle these cases. *Value based processing* aims at aggregating data to higher representation levels based on monitoring data value to provide data richness. As an example, a simple form can be unit conversion, e.g., km to mile. More generally, it aggregates different sources to produce a new data dimension, e.g., $speed = distance/time$. *QoI based processing* aims at filtering out irrelevant data in order to guarantee certain levels of QoI as requested by clients. For example, data from two sources should be filtered out to remove incoherent tuples before delivering results to applications. Figure 3.17 depicts a temporal filter of *<age, coherency>=<2 minutes, 1/2 minute>* that uses a sliding window to select the first coherent tuple of two sources.
In both cases, the views are fed by data buffers and multiple clients can share their mutual data sources while having their own mode of delivery, pull, push or even multicast push mode.

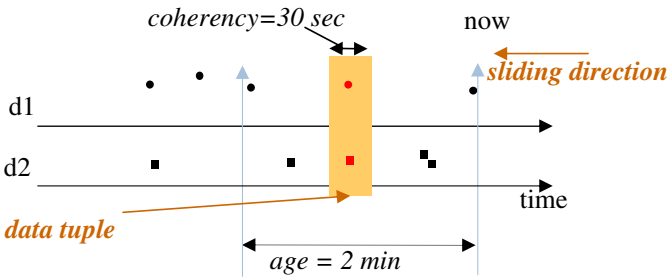


Figure 3.17: Example of temporal filter (extracted from [LDCMR10]).

QoI Control. At runtime, the *QoI control* component is used to find a configuration of the monitoring service satisfying QoI requirements and resource constraints set by the administrator. Three distinct tasks are associated to this component. First, it gathers inputs to feed the QoI control algorithms defined in the underlying model of section 3.3.2. These inputs consist of knowledge from the query repository (the current set of QoI-aware queries Q, the subset of data sources S_Q used by Q), and resource settings specified by an administrator (the set of resource constraints $\mathcal{C}_R$). Resource constraints are usually bandwidth consumption but in other contexts, it might include other resource dimensions such as CPU, memory, or battery. On the other hand data source constraints specify sampling frequency domains, which express possible configuration settings of data sources. In both

cases, we consider a constraint resolution approach in which the *QoI control* component implements a constraint resolution algorithm.
Consequently, in the second step, the component executes the QoI control algorithm to find the data source configuration C_{S_Q} satisfying the current set of queries Q under the resource constraints C_R. This algorithm can be changed at runtime thanks to dynamic reconfiguration of components (see section 3.3.5 on the framework implementation). Finally, it delivers C_{S_Q} to the *data inquiry* component, in charge of applying dynamically this new configuration into the monitoring system (see above).
The configuration of QoI control is typically executed when a new query is submitted, but executing this algorithm on demand is potentially costly. To tackle this issue, ADAMO proposes two strategies for the administrator. The first one consists in providing two reconfiguration modes: reconfigure all data sources or reconfigure only inactive data sources. The second allows for specifying a *QoI control execution policy*, which states when reconfigurations are effectively run. The default policy executes the algorithm when a change occurs. A time interval based policy allows for specifying the minimal interval between two consecutive executions, e.g., every 5 minutes. A query unit based policy defines the minimal number of changes (subscription, removal) on data queries between two consecutive executions, e.g., every 3 query updates. A last policy combines the two previous ones and avoids query pending by using the time interval.

3.3.5 Implementation

The implementation of the ADAMO framework is mainly split in two parts. The first part deals with descriptions and inputs from the different roles in the framework, consisting of QoI-aware data queries, data source and resource definitions. Since they are structural descriptions from external actors, we took advantages of code generation and extension mechanism of the Eclipse Modeling Framework (EMF) [Ecl10] to implement this part. XSD models were thus defined for QoI-aware data query, data source and resource. These models were imported into EMF tools, which generate classes that can manipulate the XML documents conformed to corresponding XML Schema, i.e. QoI-aware data query, data sources, and resource definitions.
The second part deals with core mechanisms of monitoring organized in a component-based architecture. This part has been implemented to a large extent on top of COSMOS [CRS07], a probe framework for managing context data in ubiquitous applications. This enables ADAMO to easily reuse many data sources through dedicated wrappers, which are also easy to write or to partly generate. As for its component model, ADAMO relies on the *Fractal* [BCL+06] component model, so that features like hierarchical decomposition and dynamic reconfigurations are exploited.

3.3.6 Patterns and Extension Points

In the ADAMO framework, software architects can directly used the different elements previously described to build monitoring systems. In complement of its component-based architecture, ADAMO relies on several design patterns [GHJV94] to ensure its consistency and facilitate its implementation and extension. Moreover software architects can also extend the framework by providing new versions of QoI-aware data queries, data sources or resource definitions. To do so, several extension points are provided in ADAMO.
Every monitoring system instantiated from ADAMO should compose provided components or integrate new ones implemented by extending the abstraction mechanisms At the highest level, these components must be consistent with each other and an *Abstract Factory* pattern is then used to ensure this consistency constraint. For example, to integrate a new QoI concept as a first-class constraint,

such as data precision of the query results, the *Abstract Factory* allows for ensuring that this concept is integrated all concerned monitoring entities, i.e. the *query analyzer*, *data inquiry*, *view* and *QoI control* components. As the monitoring service creates some runtime specific components, the *query analyzer* component acts a *Builder* for all views. Besides *View* elements are using the *Composite* pattern. They are implemented by *Fractal* hierarchical components, themselves extended by the COSMOS probe system [CRS07].
As multiple clients may be interested in the same source, data transmission is improved by creating a single transmission channel between ADAMO and every needed data source. While the *QoI Control* component configures the mutual data inquiry to satisfy different requests, the *Flyweight* pattern is used so that, when the view are composed, data buffers are transparently shared between clients. Finally, the *Singleton* pattern ensures that each ADAMO instance has only one *query analyzer*, *query repository* and *QoI control*. Being the front-end to clients, the *Query analyzer* acts as a *Facade* pattern.
A notable extension point in ADAMO concerns the integration of new forms of QoI control. The resource unconstrained and resource constrained systems are an obvious example in which the QoI constraints are defined differently from different inputs. Extending a QoI control thus involves both the QoI control algorithm and the inputs defined by different actors including QoI requirements, resource constraints as well as data source constraints. Since the implementation of the data query definition relies on a XML Schema, it can be extended to define new QoI requirement formalism using EMF code generation techniques (cf. section 3.3.5). Currently, ADAMO supports value based (2) or domain based ($[2..5]$) constraints for data sources, as well as value based (0.4) and relative (15%) constraints for resources. In the same way as for QoI requirements, new forms of data source constraint or resource constraint can also be defined by extending the corresponding element. Several interfaces of the framework can also be extended to specialize data inquiry and data processing. The new components implementing these interfaces have then to be registered in the framework, e.g., by providing an specific *Factory*.
Regarding the QoIcontrol algorithm, the ADAMO architecture provides two extension points, on the QoI control abstraction and on the used constraint solving back-end. The QoI control component can be exchanged with different implementations depending on when and how the QoI is effectively controlled. To do so, the component is using a QoI enforcement algorithm, following the *Strategy* pattern implemented by an abstract interface *Constraint Solver*. For example, in the resource constraint case discussed above, the algorithm formulates the problem as a *Constraint Optimization Problem* (COP), which realization implements this interface. Similarly, in the unconstrained case, the same interface is implemented to formulate the problem as *Constraint Satisfaction Problem* (CSP). In both cases the gecode[21] toolkit was used and specific code was produced by traversing and reasoning on all inputs using both the *Interpreter* and *Visitor* patterns.

3.3.7 Self-Adaptive Capability

The self-adaptive part of the ADAMO framework aims at dynamically adjusting the monitoring components according to available resources and resource constraints imposed by the system administrator. To do so ADAMO continuously monitors resource levels of the underlying system, analyzes them and takes the correct action in order to enforce resource constraints. This self-adaptive capability has been implemented through a general feedback control loop [BDG+09] which is deployed over the QoI enforcement component in the case of a resource constrained system. The loop is decom-

[21] http://www.gecode.org/

posed in three activities, *monitoring*, *decision* and *action*, following a simplified form of autonomic loop [KC03].

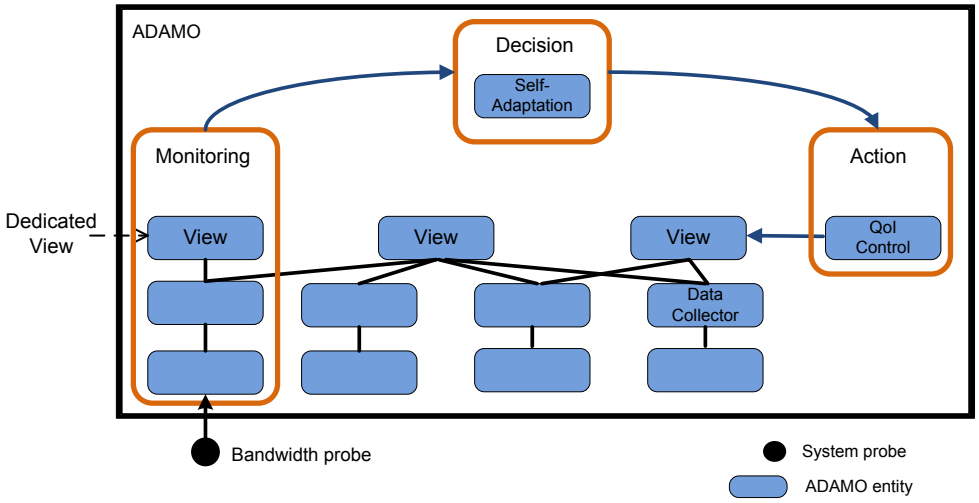


Figure 3.18: ADAMO self-adaptive loop.

An interesting property of the architecture of this feedback control loop is that it reuses many elements of the ADAMO framework itself. Consequently, it can be seen as a case study demonstrating its extensibility.

Figure 3.18 depicts the loop activities and their integration with different ADAMO elements. First, we take advantage of ADAMO's features to monitor the necessary resource dimensions of the runtime support. Only dimensions that concern resource constraints are monitored[22]. As the collection of such resources is generally relatively negligible with regards to other resources used in the functional data collection, the QoI control component with a default configuration is used. In this context, the decision part of the loop plays the role of client while rest of the ADAMO framework has the data provider role.

To make its decision, the self-adaptation component takes as inputs the resource constraints and resource information collected from the dedicated data collector. It notifies the QoI control whenever the resource levels are considered to be changed, either through a degradation or elevation of some resource dimension. The default decision making technique uses a simple threshold to compare with available resource. This is obviously a basic solution that has many drawbacks, but on the other hand, the architecture of the loop is explicit enough to implement other techniques, such as prediction based on historical resource information.

On the action side, the resource changes bring about violation of resource constraints or resource under-performance. The QoI control component is thus reused to enforce monitoring constraints, leading typically to the trade-off between QoI and available resources. Since runtime issues of QoI control are also concerned, the principle of the QoI control policy (cf. Section 3.3.4) is applied to the self-adaptation cycle. However, reconfiguration caused by applicative clients (i.e. submission or removal of a query) and self-adaptation are distinguished so that two QoI control policies coexist. Besides it is the default policy, applying the QoI control on each change, which is used on the self-

[22]We suppose that the underlying system can provide necessary data sources to collect resource information, such as bandwidth consumption or CPU usage

adaptive loop.

3.3.8 Related Work

Context-aware systems are typically concerned with QoI to perceive situations and adapt applications based on the recognized context. Quality of Context is well studied in [BKS03, MTD08, AHAES08] where many dimensions are proposed, including precision, freshness and consistency of the monitored data. These works, however, do not address the architecture of context-aware systems or the problem of maximizing QoI over a set of constraints. Among the different works on context-aware management systems, COSMOS [CRS07] is an architecture based on components (called context nodes) that are responsible to produce higher level context information from data gathered at lower architectural layers. Several composition patterns of these components allows for architecting the context-aware part of applications. In [ACC09], COSMOS is extended to support Quality of Context (QoC) by using a specialized component to filter and evaluate QoC from collected information. Although they do not address the problem of maximizing QoC in overloaded situations their architecture is highly modular and extensible, and allows for introducing controlled trade-offs between QoI requirements and resource consumption.
Poladian et al. [PSGS04, PGS+07] focus on adaptive systems based on multiple concurrent applications running on local computing devices with limited memory, CPU and bandwidth. They propose an analytical model and an efficient algorithm to decide how to allocate scarce resources to applications, and how to set the quality parameters of each application to best satisfy user and supplier preferences. Their approach could fit well into the ADAMO framework in order to adjust the monitoring to current conditions, given QoI objectives.
Several works propose sophisticated algorithms and optimization techniques in the domain of data stream processing systems. They are concerned with the problem of saving network or compute resource to deliver accurate information. For example some proposals drop data tuples to reduce bandwidth and processing when needed [BDM04, TcZ07]. Load shedding is then formalized as an optimization problem with the goal of minimizing query inaccuracy within the limits imposed by resource constraints. However these works does not address the design of the monitoring framework to implement them in a modular and flexible manner.
Some others works aim at predicting runtime malfunctions in software systems. In [MW06, MJW08] the authors propose a new approach to monitor multi-tier transaction systems. It uses relationships between the monitored data in the form of regression models to determine normal operation, with minimal monitoring, from areas that need more monitoring in the event of anomalies. Nevertheless, they not consider QoI requirements such as the age of monitoring data.

3.3.9 Summary

We have described, ADAMO, an adaptive monitoring framework that tackles different QoI-aware data queries over dynamic data streams. ADAMO provides solutions for i) flexible access to dynamic data streams for multiple clients with different QoI needs, ii) taking into account QoI constraints to generate and configure appropriate elements in the monitoring system, iii) making the monitoring system adaptable to resource constraints at runtime, and iv) managing data queries in a static or incremental way. The proposed system relies on a constraint-solving approach to transform QoI needs into probe configuration settings. A QoI control component implements this behavior and is open to integrate different QoI control algorithms. A default static configuration is possible, but two adaptive configurations are more interesting. The QoI enforcement on a resource unconstrained system aims

at minimizing monitoring resource consumption (limited to bandwidth) while guaranteeing all client QoI needs. In a resource constrained system the QoI enforcement seeks the trade-off between QoI and available resources to maximize clients' utility. ADAMO also provides a self-adaptive capability to best satisfy QoI requirements. A feedback control loop enables the framework to continuously monitor the system resources and dynamically enforce monitoring constraints.
As a framework, several extension points enable software architects to customize or integrate new features including data processors and QoI control mechanisms. Especially, the QoI control component provides an abstract constraint solver generator that facilitates the integration of new solvers. The usage of several design patterns and EMF-based generative techniques improves consistency and facilitates extension of the framework. Two variants of a versatile crisis management system have been developed (fire fighting, flood management) and served as case studies. The fire fighting case study have been integrated in a end-to-end demonstration in the context of the ANR Semeuse project[23]. Measurements on the instantiated monitoring systems also show that the adaptive monitoring properties have been met with good performance and resource efficiency.

[23] http://www.semeuse.org

CHAPTER 4

Feature Modeling

Contents

4.1 Supporting Separation of Concerns for FM Management 92
4.1.1 Background on Feature Models 92
4.1.2 Motivations 93
4.1.3 Overview and Semantic Basis 94
4.1.4 Composing FM 95
4.1.5 Decomposing FM 98
4.1.6 Implementation, Complexity and Performance 99
4.1.7 Related Work 100
4.1.8 Summary 101
4.2 A Domain-Specific Language for Large Scale Management of FM 101
4.2.1 Motivations 101
4.2.2 Rationale for an FM Management DSL 102
4.2.3 FAMILIAR 103
4.2.4 Environment and Reasoning Back-end 108
4.2.5 Summary 109
4.3 Applications of SoC in Feature Modeling 109
4.3.1 Composing Multiple Variability Artifacts to Assemble Coherent Workflows 110
4.3.2 Modeling Variability From Requirements to Runtime in Video Surveillance Systems 112
4.3.3 Reverse Engineering Architectural Variability 113
4.3.4 Management of Product Line Variability and Software Variability 115
4.3.5 Summary 116

This chapter presents our research work on large scale management of feature models which have been conducted from 2008 on.

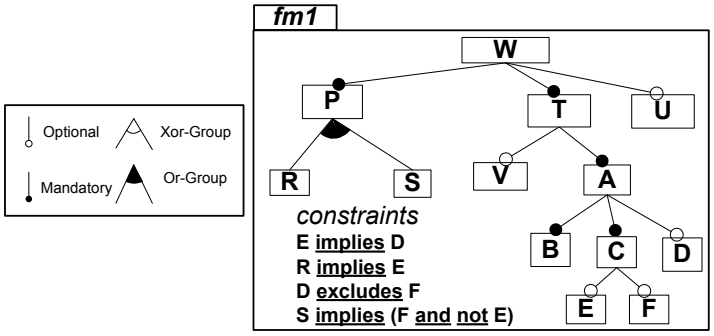
Making the analogy of other industries such as automotive or semiconductor sectors, Software Product Line (SPL) engineering is a paradigm shift towards modeling and developing software system families rather than individual systems [CN01]. Its goal is to produce a family of related program variants for a domain [PBvdL05]. SPL development starts with an analysis of the domain to identify commonalities and differences between the members of the product line. A common way is to describe variabilities of an SPL in terms of features which are domain abstractions relevant to stakeholders and are typically increments in program functionality [AK09]. *Feature Models* (FMs) [KKL+98] are now widely used to compactly define all features and their valid combinations in an SPL. As FMs are getting increasingly large and complex, our work focused on applying the principles of *separation of concerns (SoC)* so to provide composition and decomposition for FMs. For a better support, a dedicated language was developed and several significant case studies were developed to validate these contributions.

4.1 Supporting Separation of Concerns for FM Management

This section shares material with the SLE'09 paper "Composing Feature Models" [ACLF09] and the ASE'11 paper "Slicing Feature Models" [ACLF11c]. It concerns Mathieu Acher's PhD Thesis, co-supervised with Philippe Lahire, and a collaborative work with Robert B. France.

4.1.1 Background on Feature Models

Feature Models (FMs) were first introduced in the FODA method [KKL+98], which also provided a graphical representation through Feature Diagram. FMs are now widely adopted with support of formal semantics, reasoning techniques and tooling [SHTB07, CW07, AK09, BSRC10]. An FM defines both a *hierarchy*, which structures features into levels of increasing details, and some *variability* aspects expressed through several mechanisms. When decomposing a feature into subfeatures, the subfeatures may be *optional* or *mandatory* or may form *Xor* or *Or*-groups. In addition, any *propositional* constraints (e.g., implies or excludes) can be specified to express more complex dependencies between features. We consider that an FM is composed of a feature diagram coupled with a set of constraints expressed in propositional logic. Figure 4.1a shows an example of an FM. The feature diagram is depicted using a FODA-like graphical notation used throughout this chapter.



(a) FM: FODA-like representation

$$
\begin{aligned}
&[[fm1]] = \{\\
&\{A, B, C, D, E, P, R, T, U, W\},\\
&\{A, B, C, F, P, S, T, U, W\},\\
&\{A, B, C, D, E, P, R, T, W\},\\
&\{A, B, C, F, P, S, T, V, W\},\\
&\{A, B, C, F, P, S, T, U, V, W\},\\
&\{A, B, C, F, P, S, T, W\},\\
&\{A, B, C, D, E, P, R, T, V, W\},\\
&\{A, B, C, D, E, P, R, T, U, V, W\}\\
&\}
\end{aligned}
$$

(b) corresponding set of configurations

$$
\begin{aligned}
&\phi_{fm_1} = W \text{ // root}\\
&\wedge\ W \Leftrightarrow P \wedge\ W \Leftrightarrow T\\
&\wedge\ U \Rightarrow W \text{ // optional}\\
&\wedge\ V \Rightarrow T\\
&\wedge\ A \Leftrightarrow T \text{ // mandatory}\\
&\wedge\ A \Leftrightarrow B \wedge\ A \Leftrightarrow C\\
&\wedge\ D \Rightarrow A\\
&\wedge\ E \Rightarrow C \wedge\ F \Rightarrow C\\
&\text{// Or-group}\\
&\wedge\ P \Rightarrow R \vee S\\
&\wedge\ R \Rightarrow P \wedge\ S \Rightarrow P\\
&\text{// constraints}\\
&\wedge\ E \Rightarrow D \wedge\ R \Rightarrow E\\
&\wedge\ D \Rightarrow \neg F \wedge\ S \Rightarrow (F \wedge \neg E)
\end{aligned}
$$

(c) corresponding propositional formula

Figure 4.1: FM, set of configurations and propositional logic encoding.

The *hierarchy* of an FM is represented by a rooted tree $G = (\mathcal{F}, E, r)$ where $\mathcal{F}$ is a finite set of features and $E \subseteq \mathcal{F} \times \mathcal{F}$ is a finite set of edges (edges represent top-down hierarchical decomposition

of features, i.e., parent-child relations between them) ; $r \in \mathcal{F}$ being the root feature.
An FM defines a set of valid feature *configurations*. A valid configuration is obtained by selecting features so that *i)* if a feature is selected, its parent is also selected; *ii)* if a parent is selected, all the mandatory subfeatures, exactly one subfeature in each of its Xor-groups, and at least one of its Or groups are selected; *iii)* propositional constraints hold. For example, in Figure 4.1a, D while E cannot be selected at the same time, E cannot be selected without C due to the parent-child relation between them.
We define a *configuration* of a feature model FM as a set of selected features. $[\![FM]\!]$ denotes the set of valid configurations of the feature model FM and is thus a set of sets of features (cf. Figure 4.1b). FMs have been semantically related to propositional logic [CW07]. The set of configurations represented by an FM can be described by a propositional formula ϕ defined over a set of Boolean variables, where each variable corresponds to a feature (cf. Figure 4.1c).The translation of FMs into logic representations allows for using reasoning techniques for automated FM analyses [MM09, BSRC10].

4.1.2 Motivations

FMs are becoming increasingly complex. There are a number of factors that contribute to their growing complexity. A first factor is that FMs are being used not only to describe variability in software designs, but also variability in wider system contexts [MPH+07, HT08, TBC+09]. For example, features may refer to high-level requirements as well as to properties of the software platform. Similarly, external variability, visible to customers, can be distinguished from internal variability, hidden from customers, in an FM [PBvdL05].
Another contributing factor is the use of multiple FMs to describe SPLs. It has been observed that maintaining a single large FM for the entire system may not be feasible [KKL+98, MPH+07, PBvdL05, DGRN10]. Following a model-based approach, several FMs are usually designed to describe software features at various levels of abstraction and to manage variability of artifacts that are produced in different development phases [PBvdL05, AK09]. In addition, organizations are increasingly faced with the challenge of managing variability in product parts provided by external suppliers [HT08, HTM09]. The need to support multiple SPLs (also called product populations) makes developing SPLs challenging [PBvdL05]. Product populations with FMs consisting of hundreds to thousands of features have been observed [RW07, MC10, BSRC10]. Finally, automated extraction of FMs from large implemented software systems [SLB+11], can produce FMs with thousands of features.
With FMs being handled by several stakeholders, or even different organizations, managing them with a large number of features that are related in a variety of ways is intuitively a problem of *Separation of Concerns* (SoC). The sought benefits are indeed similar to the ones of software engineering disciplines, i.e., reduced complexity, improved reusability and simpler evolution [TOHS99]. As with any formalism, SoC techniques may differ depending on the supported kinds of decomposition and composition. In feature modeling, previous works have mainly focused in providing techniques to reason on relevant subparts of a larger FM, either by enabling staged configuration by different stakeholders with FM specialization [CHE05b] or by allowing view extraction from an FM [HHS10]. These approaches make the assumption that a single FM keeps all the information, which is not acceptable any more for very large FMs recently observed.
We advocated that some fully-fledged SoC support is needed for large scale FM management and that this support should be based on a set of decomposition and composition operators. Moreover all activities related to FMs involve reasoning about the represented configurations and the impact on the described variable artifacts. The provided form of composition and decomposition must thus

ensure salient properties on them so that automatic reasoning is made possible [BSRC10]. These requirements have been confirmed on a large scale, with experience in the development of a real-world Video Surveillance Systems SPL [ACL+11] and a Medical Imaging Workflow SPL [ACLF10b], in which several variability concerns are to be managed (see section 4.3).

4.1.3 Overview and Semantic Basis

We identify three composition operators and one decomposition operator.
The first identified mechanism, called *insert*, aims at introducing new features, already organized in a feature model, into a specific location of another existing feature model. It is primarily used to populate an existing feature model with additional information, but can also allow for reusing an existing feature model when creating a larger composed feature model.
The second identified mechanism, called *aggregate*, supports cross-tree constraints between features so that separated feature models can be inter-related.
The third mechanism, called *merge*, is dedicated to the composition of feature models that have similar features[24].
The proposed decomposition operator, called *slice*, goes beyond a simple extraction of features and automatically produces a feature model that contains only the relevant subset of features according to a slicing criterion.
To determine the underlying semantics from these intuitive definition, we consider that the primary meaning of a feature model, known as its *configuration semantics*, is a set of legal configurations, i.e., sets of selected features that respect the dependencies entailed by the diagram and the cross-tree constraints (see section 4.1.1). We thus define the semantics properties of each operator in terms of the relationship between the configuration sets of the input models and the resulting feature model. In particular, we rely on the classification proposed in [TBK09] that covers all the changes a designer can produce on a feature model and that provides a sound basis for reasoning about these changes. In [TBK09], the authors distinguish and classify four feature model adaptations that we detail now. Let f and g be two feature models, $[\![f]\!]$ and $[\![g]\!]$ denote their respective sets of configurations.

- ◇ f is a *specialization* of g if $[\![f]\!] \subset [\![g]\!]$
- ◇ f is a *generalization* of g if $[\![g]\!] \subset [\![f]\!]$
- ◇ f is a *refactoring* of g if $[\![g]\!] = [\![f]\!]$
- ◇ f is an *arbitrary edit* of g if f is neither a specialization, a generalization nor a refactoring of g.

The term "edits" is used as a set of changes to a feature model. An example of edit given in [TBK09] is "moving a feature from one branch to another".
Feature models have other important properties that can be extracted by automated techniques and that will be relevant to our proposals. In particular, a feature model may represent no valid configuration, it is then a void feature model.
A feature f of FM is dead if it cannot be part of any of the valid configurations of FM. The set of dead features of FM is noted $deads(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \notin c\}$.
A feature f of FM is a core feature if it is part of all valid configurations of FM. The set of core features of FM is noted $cores(FM) = \{f \in \mathcal{F} \mid \forall c \in [\![FM]\!], f \in c\}$.
Furthermore, we encode the set of configurations as a *propositional formula* and consider the feature hierarchy.

[24] We consider that feature names are unique in all considered feature models.

Another important property of an FM is indeed the way features are organized, directly reflected in the feature hierarchy. We recall that two FMs can have identical configuration semantics, yet different hierarchies and thus meaning [TBK09, BSRC10]. Consequently, we consider that the feature hierarchy should also be part of the semantics of the operators.

4.1.4 Composing FM

Insert

The insert operator aims at introducing elements from a given feature model in another one. To define the behavior and semantics of this operator, we borrow the terminology of aspects, calling one FM the base model and the one inserting elements in it, the aspect model. We define insert as being a versatile operator to add features, corresponding to the elements to be inserted, without any merging. The precondition of the insert operator requires that the intersection between the set of features of the base feature model and the one of the aspect feature model is empty. This condition preserves the well-formed property of the composed feature model which states that each feature's name is unique. Then, a feature of the base feature model must act as a *joint point* where the aspect feature model will be inserted, and there will be various ways to insert an aspect feature model according some variability operators (optional, mandatory, etc.). To materialize it, we syntactically define the insert operator as follows:

```
insert  (aFM: FeatureModel, bFM: FeatureModel, jptFeature: Feature, vop: VariabilityOperator)
```

It takes four arguments: the aspect feature model to be inserted (aFM), the base feature model bFM, the targeted feature (a feature in the base model) where the insertion is to be done ($jptFeature$), and the variability operator vop (whose value is either *Mandatory*, *Optional*, *Xor*, *Or*).
When the variability operator is either *Mandatory* or *Optional*, the root of the aspect feature model is inserted as a *child* feature of the join point feature. When the variability operator is either *Xor* or *Or*, the root of the aspect feature model is inserted as a *sibling* feature of the join point feature. An insertion may also create a feature group if needed.
How an aspect feature model is inserted has a direct impact on the set of configurations of the resulting feature model. Intuitively, it is tempting to state that when an aspect feature model is added somewhere in a base feature model $Base_{FM}$, the set of configurations of $Base_{FM}$ necessarily *grows* , that is, new configurations are added to the original set of configuration (and thereby $Base'_{FM}$ is a *generalization* of $Base_{FM}$). We have demonstrated that it is not the case and that the kind of relationship between $Base_{FM}$ and $Base'_{FM}$ is dependent both on the original properties of $Base_{FM}$, the variability operator and the joinpoint feature chosen. Importantly, $Base'_{FM}$ being a specialization of $Base_{FM}$ is not possible. Details and proofs related to all cases can be found in Mathieu Acher's PhD Thesis [Ach11].

Aggregate

The *aggregate* operator supports cross-tree constraints between features so that separated FMs can be inter-related. Features in input feature models are related to each other through relations expressed in propositional logic. Here we do not make the distinction between a base feature model and an aspect feature model, the features models are equally important.
We syntactically define the aggregate operator as follows:

```
aggregate (sFM: set of FeatureModel, sCst: set of  Constraint)
```

The aggregate operator takes as input a set of feature models (sFM), a set of propositional constraints ($sCst$) and produces a new feature model. The input FMs are aggregated under a synthetic root $synthetic_{ft}$[25] so that the root features of input FMs are child-mandatory features of $synthetic_{ft}$. This ensures that no arbitrary feature has to be chosen to be a root, leading to meaningless results. In addition, the propositional constraints are added in the resulting FM.
The properties of the aggregated FM heavily depends on the set of propositional constraints used during the aggregate. It may lead to situations where the aggregated FM does not represent any valid configuration or include dead or core features. We consider that the aggregate operator is purely syntactical. Semantics properties are more relevant when merging feature models, as shown below.

Merge

The *merge* operator is dedicated to the composition of FMs that exhibit similar features, i.e. features with the same name. In this case, the merge operator can be used to *merge* the overlapping parts of the FMs and then to obtain an integrated FM. The merge uses name-based matching: two features match if and only if they have the same name.
We syntactically define the aggregate operator as follows:

```
merge (sFM: set of FeatureModel, mode: MergeMode)
```

We consider that the merge operator takes as input a set of feature models sFM, a merging mode $mode$ and produces a new feature model. As there are different ways to merge two or more than two feature models, several modes are defined for the merge operator. The merging mode can be either *union*, *strict union*, *intersection* or *diff*. The set of configurations expressed by the merge feature model depends on this merging mode.
The *union* mode is the most inclusive option. The merged FM includes all the valid configuration defined by the input FMs and is defined as follows:

$$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![Result]\!]$$

This property is obviously too loose to be usable in the general case. In particular, some valid configurations of FM_r are neither valid in FM_1 nor in FM_2, being open to different interpretations.
On the other hand, the merge operator in the strict union mode is denoted $FM_1 \oplus_{\cup} FM_2 = Result$ and provides some interesting properties. We use "strict" as we just want to obtain a merged feature model FM_r that represents exactly the union of the two sets of configurations of FM_1 and FM_2. In this mode, each valid configuration of FM_r is also valid either in FM_1 or FM_2 (or in both):

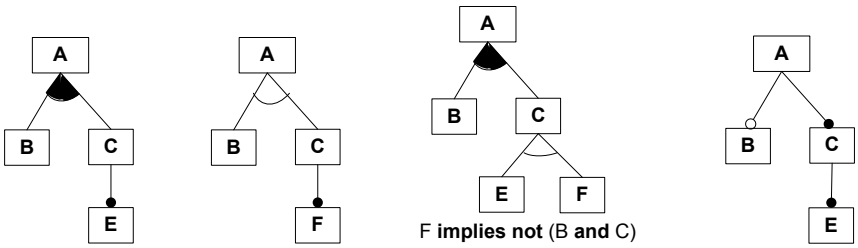
$$[\![FM_1]\!] \cup [\![FM_2]\!] = [\![FM_r]\!]$$

An example is given in given in Figure 4.2: fm_{m56} (see Figure 4.2c) is the feature model resulting from the merge in strict union mode of fm_{m5} (see Figure 4.2a) and fm_{m6} (see Figure 4.2b).
Another merge operator, called *diff*, is denoted as $FM_1 \oplus_{\setminus} FM_2 = Result$. The following defines the semantics of this operator:

$$[\![FM_1]\!] \setminus [\![FM_2]\!] = \{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![Result]\!]$$

An example is given in given in Figure 4.2: fm_{diff56} (see Figure 4.2d) is the feature model resulting from the merge in diff mode of fm_{m5} and fm_{m6}.

[25] A synthetic root is a fake elements added at the root of a FM. It is only used to ensure the well-formedness of the hierarchy and is *not* part of the set of configurations of the aggregated feature model.



(a) fm_{m5} (b) fm_{m6} (c) fm_{m56} (d) fm_{diff56}

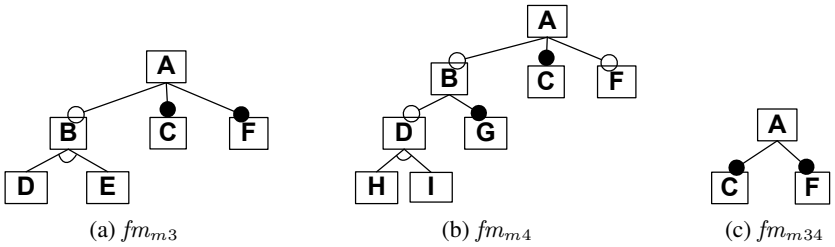
$$fm_{m5} \oplus_{\cup} fm_{m6} = fm_{m56}$$
$$fm_{m5} \oplus_{\setminus} fm_{m6} = fm_{diff56}$$

Figure 4.2: Merging in strict union and diff modes.

The *intersection* mode is the most restrictive option: the merged FM, FM_r, expresses the common valid configurations of FM_1 and FM_2. The merge operator in the intersection mode is denoted as follows: $FM_1 \oplus_{\cap} FM_2 = Result$. The relationship between a merged FM $Result$ in intersection mode and two input FMs FM_1 and FM_2 can be expressed as follows:

$$[\![FM_1]\!] \cap [\![FM_2]\!] = [\![Result]\!]$$

An example is given in Figure 4.3.



(a) fm_{m3} (b) fm_{m4} (c) fm_{m34}

$$fm_{m3} \oplus_{\cap} fm_{m4} = fm_{m34}$$

Figure 4.3: Merging in intersection mode.

We encode each input FMs involved in the merging operation as propositional formulas and, depending on the merging mode, we apply some Boolean operations over these formulas. For instance, the strict union of two sets of configurations represented by two FMs, FM_1, and FM_2, can be computed as follows. First, FM_1 (resp. FM_2) FMs are encoded into a propositional formula ϕ_{FM_1} (resp. ϕ_{FM_2}). Then, the following formula is obtained:
$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \vee (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$
where $\mathcal{F}_{FM_1}$ (resp. $\mathcal{F}_{FM_2}$) is the set of features of FM_1 (resp. FM_2) and, $\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1}$ denotes the complement (or difference) of $\mathcal{F}_{FM_2}$ with respect to $\mathcal{F}_{FM_1}$; not is a function that, given a non-empty set of features, returns the Boolean conjunction of all negated variables corresponding to features:
$not(\{f_1, f_2, ..., f_n\}) = \bigwedge_{i=1..n} \neg f_i$
Computing the intersection of two sets of configurations represented by two FMs, FM_1, and FM_2, follows the same principles and we obtain:

$\phi_{Result} = (\phi_{FM_1} \wedge not(\mathcal{F}_{FM_2} \setminus \mathcal{F}_{FM_1})) \wedge (\phi_{FM_2} \wedge not(\mathcal{F}_{FM_1} \setminus \mathcal{F}_{FM_2}))$
Several FMs, with different hierarchies, can represent the same set of configurations [TBK09, BSRC10]. So in particular several merged FMs can be produced and consistently represent the expected set of configurations while having different hierarchies. The problem of choosing a hierarchy from amongst a set of hierarchies can be formulated as a minimum spanning tree (MST) problem. We consider a connected, undirected, weighted graph $G_m = (V_m, E_m)$ that includes all features and edges[26] of the input FMs and assigns to each edge $e \in E_m$ a weight equal to the $-n$, n being the number of times e occurs in the different hierarchies of the input FMs. Then the choice of the hierarchy is a minimum spanning tree of G_m.

4.1.5 Decomposing FM

The overall idea behind FM slicing is similar to program slicing [Wei81]. Program slicing has been successfully applied in computer programming: It has several practical applications in program understanding, maintenance, debugging, differencing, merging, etc. It aims at simplifying or abstracting programs by focusing on selected aspects of semantics. Program slicing techniques proceed in two steps: the subset of elements of interest (e.g., a set of variables of interest and a program location), called the slicing *criterion*, is first identified ; then, a *slice* (e.g., a subset of the source code) is computed. In the context of FMs, we define the slicing criterion as a set of features considered to be relevant by an SPL practitioner while the obtained slice is a new FM.
We syntactically define the slice operator as follows:

```
slice (aFm: FeatureModel, criterion: set of Feature)
```

The result of the slicing operation is a new FM, FM_{slice}, such that:
$[\![FM_{slice}]\!] = \{ x \in [\![FM]\!] \mid x \cap \mathcal{F}_{slice} \}$ (called the *projected* set of configurations);
For a slicing $FM_{slice} = \Pi_{ft_1,ft_2,...,ft_n}(FM)$, the propositional formula ϕ_{slice} corresponding to FM_{slice} can be defined as follows:

$$\phi_{slice} \equiv \exists\, ftx_1, ftx_2, \ldots ftx_{m'}\ \phi$$

where $ftx_1, ftx_2, \ldots ftx_{m'} \in (\mathcal{F} \setminus \mathcal{F}_{slice}) = \mathcal{F}_{removed}$.
ϕ_{slice} is obtained from ϕ by *existentially quantifying out* variables in $\mathcal{F}_{removed}$. Intuitively, all occurrences of features that are not present in any configuration of FM_{slice} are removed by existential quantification in ϕ.
The hierarchy of the sliced FM is defined as follows: $G_{slice} = (\mathcal{F}_{FM_{slice}}, E_{slice})$ with
$\mathcal{F}_{FM_{slice}} = ((\mathcal{F}_{slice} \setminus deads(FM)) \cup synthetic_s)$ and $E_{slice} \subseteq E$ such that
$E_{slice} = \{e = (v, v') \mid e \in E' \wedge \nexists\, v'' \in E' : ((v, v'') \in E' \wedge (v', v'') \in E')\}$
where $G' = (\mathcal{F}', E')$ is the transitive closure of G_{slice} ;
Intuitively, some features have to be connected to their closest ancestor if their parent is not part of the sliced FM. We consider that the synthetic root is *not* part of $[\![FM_{slice}]\!]$ and is only here to ensure the well-formedness of the hierarchy (if needs be). The synthetic root can be removed from G_{slice} when one or more than one of its child feature is a *core* feature, otherwise the synthetic root is necessary (e.g., for the purpose of visualization). In the case there is exactly one core child feature f_{core}, the root feature of FM_{slice} becomes f_{core}. In case the synthetic root has two or more than two child features that are core features, a procedure should choose one.

[26] In intersection mode, features (and associated edges) known to be not included in the merged FM are removed.

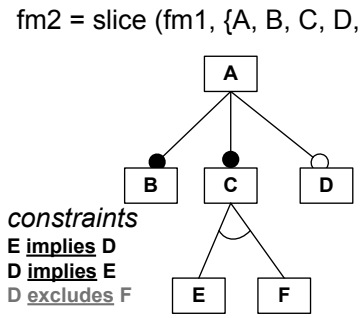


Figure 4.4: Example of slice applied on the feature model of Figure. 4.1a.

An example is given in Figure 4.4 where the slicing criterion corresponds to the set of features A, B, C, D, E and F. The hierarchy of fm_2 does not alter the structure (i.e., parent-child relationships) of the original feature model fm_1. It corresponds to a subtree of the tree of fm_1 whose root is feature A. The valid configurations characterized by fm_2 corresponds to the valid configurations of the original feature model fm_1, when looking only at the specific features of the criterion. It can be seen as a *projection* of the relational algebra on $[\![fm1]\!]$ (see Figure 4.1b) when the features not included in the criterion (W, P, ..., U) are discarded. The variability of fm_2 is then set to accurately represent $[\![fm_2]\!]$. I can be first observed that features E and F form an Xor-group in fm_2 whereas they are optional features in fm_1. Actually they were already mutually exclusive in fm_1. A new constraint (D implies E) has been added to fm_2, as it is logically entailed by fm_1. Finally the constraint D excludes F is not added to fm_2 since it is redundant (i.e., does not alter $[\![fm2]\!]$).

4.1.6 Implementation, Complexity and Performance

The implementation of the aggregate operator is without any difficulty. The implementations of the merge and slice operators present much more difficulties but rely on the same principles: we first *i)* compute the propositional formula representing the expected set of configurations and then *ii)* we apply propositional logic reasoning techniques to construct an FM (including its hierarchy, variability information and cross-tree constraints) from the propositional formula.
We reuse and adapt techniques presented in [CW07, SLB+11] where the authors construct a feature diagram from a propositional formula. The technique notably allows one to detect *Xor-* or *Or-* groups (using the method of prime implicants). A major difference is that, in our work, we already *know* the resulting hierarchy and how features are grouped. We thus exploit this information to streamline the algorithm. Details can be found in [ACLF11c] and [Ach11].
Hierarchy computations use standard graph techniques (e.g., minimum spanning tree technique in the case of merge) and are not an issue, even for very large FMs. Currently, our handling of logical operations relies on Binary Decision Diagrams (BDDs) [BRB90]. Computing the existential quantification of BDDs can be performed in at most polynomial time with respect to the sizes of the BDDs involved [BRB90]. This property is important in the case of the slice operator, i.e., when computing ϕ_{slice}. In addition, computing the negation, conjunction or disjunction of two BDDs can be performed in at most polynomial time with respect to the sizes of the BDDs involved (these logical operations are used extensively during the merging of several FMs). As argued in [CW07], the cost of feature diagram construction is polynomial regarding the size of the BDD representing the input propositional formula (the most expensive step is the computation of prime implicants). The size of a BDD struc-

ture is highly sensitive to the variable ordering used in its construction. An inappropriate ordering may prevent the BDD encoding of an FM. We thus reuse the heuristics developed in [MWCC08] (i.e., Pre-CL-MinSpan) that are known to scale for up to 2000 features.

In a first evaluation, we used several small and medium-sized FMs that were publicly available from SPLOT [MM09] repository as well as FMs from our case studies. We performed our experiments on more than 100 FMs, the bigger one having 290 features. Computing the slice or the merged was almost instantaneous in all cases.
Another evaluation relies on randomly generated FMs following the procedure described in [MM09]. We varied *i)* the number of features, noted $\#features$, from 100 to 2000 features (the practical limits of BDD) ; *ii)* $CTCR$ (the ratio of the number of features in the constraints to the number of features in the feature hierarchy expressed as percentage) from 10% to 100%. In each generated model, each type of mandatory, optional, Xor and Or-groups was added with equal probability. The cross-tree constraints were generated as a single Random 3-CNF formula. The results first shows that the synthesis of the feature diagram has practical limits (up to 800 features), for both merge and slice. We observed that the slicing technique can scale even for an FM with 2000 features if the percentage of features to slice is $\leq 35\%$. The reason is that the size of a BDD will always be smaller or at least unchanged after existential quantification. Another result is that the primary limit of the BDD-based implementation lies in the difficulties to compile BDD from the original FM. In particular, the total number of features in the resulting merged FM should not be less than 2000 features, otherwise it is impossible to having a BDD-representation of the merged formula. For the slice operator, whenever an FM can be represented as a BDD, ϕ_{slice} can be computed. Hence the encoding of ϕ_{slice} can scale up to 2000 features with a CTCR of 10.
Recently, She et al. proposed techniques to reverse engineering very large FMs, with more than 5000 features [SLB+11]. As BDDs do not scale, the authors adapted their techniques to use SAT solvers and reported some significant improvement of scalability. We discuss possible improvements, such as a SAT-based implementation of slicing and merging, in section 5.2.5.

4.1.7 Related Work

Regarding composition, a few works [AGM+06, GMB06, SHTB07, SBRCT08] consider some forms of composition for FMs and suggest the use of a merge operator.
In [SHTB07] and [HST+08], the authors considers that that intersection or (strict) union can be realized by maintaining *separate* input feature models and inter-relating them with constraints. The limitations of this reference-based approach are that *i)* the resulting merged feature model may contain anomalies (false optional features, dead features) and *ii)* the entire set of features of input feature models is included in the resulting feature model so that the number of features quickly increases and large feature models are produced. In comparison, our resulting merged feature models are more compact and more readable.
Alves et al. motivate the need to manage the evolution of FMs (or more generally of an SPL) and extend the notion of refactoring to FMs [AGM+06, GMB06]. Although their work is focused on refactoring single FMs, they also suggest to use these rules to merge FMs. Inspired by the work of Alves et al., Segura et al. provide a catalog of visual rules to describe how to merge FMs [SBRCT08]. Our proposal goes further since we clarified the semantics of the merge, in terms of configuration semantics *and* feature hierarchy. An in-depth comparison of implementation approaches can be found in [ACLF10a].

Recently, in [vGN10], merging is realized through an algorithm that guarantees some properties (minimality, parent compatibility, commutativity, etc.) of the merged feature models, but this algorithm is less general as it assumes that input feature models are parent-compatible.

4.1.8 Summary

We described a set of composition and decomposition operators (insert, aggregate, merge, slice) dedicated to the formalism of feature model. The merge operator notably produces more compact feature models than existing techniques. It thus facilitates the handling, understanding and analysis of multiple feature models. Moreover the slice operator allows one to find semantically meaningful decompositions of a feature model. The implementation of the two operators guarantees that the set of configurations and the hierarchy of the produced feature model are consistent with a well-defined semantics. The use of propositional logic techniques for the implementation of the merge operators outperforms current solutions, raises previous limitations and notably preserves, by construction, the set of configurations.
But for these composition and decomposition operators to be fully usable at managing large scale feature models, one need to combine them with other manipulation and reasoning operations. The next section will detail the dedicated language tackling this issue. A summary of several applicative case studies complements this description in the last section.

4.2 A Domain-Specific Language for Large Scale Management of FM

This section shares material with the SAC'11 paper "A Domain-Specific Language for Managing Feature Models" [ACLF11a] and the VAMOS'11 paper "Managing Feature Models with Familiar: a Demonstration of the Language and its Tool Support" [ACLF11b]. Like the previous section, it concerns Mathieu Acher's PhD Thesis and a collaborative work with Philippe Lahire and Robert B. France.

4.2.1 Motivations

In the previous section, we presented a set of *composition* operators for FMs (insert, merge, aggregate) that preserved semantic properties expressed in terms of configuration sets of the composed FMs. The decomposition counterpart was also described. This is a *slicing* algorithm that produces a projection of an FM (a slice) using a slicing criterion.
Our experience in the development of a real-world Video Surveillance Systems SPL [ACL+11] and a Medical Imaging Workflow SPL [ACG+11] suggests that support for separating concerns and synthesizing large FMs from smaller FMs can significantly improve management of complex SPLs[27]. But providing support for FM composition is not enough. Manually analyzing complex FMs is an error-prone and tedious task, and thus there is a need for tools that automate significant aspects of the reasoning process [BSRC10]. To support effective development and management of large complex FM, SPL developers need scalable FM development environments that allow them to better control how large FMs are created, analyzed and evolved. This can be done by giving developers the means to define complex operations on FMs by combining basic FM operators that, for example, compose FMs, add new features, remove features, and support reasoning about FM properties.

[27] Section 4.3 summarizes this experience on several applicative case studies.

4.2.2 Rationale for an FM Management DSL

There are at least three possible solutions to meet the requirements above. One approach is to reuse existing FM development tools and editors. The other two involve using a language, either general-purpose or domain-specific.
Several graphical FM editors are currently available, and some do provide support for managing some aspects of FM development, for example, *pure::variants* [pur06], SPLOT [MBC09] or FeatureIDE [KTS+09]. For large scale management, *pure::variants* is a commercial tool with good support for binding FMs to other models and for code generation. *FeatureIDE* is a comprehensive environment that interconnects with different FM management tools and has a Java API to manipulate FMs. Integration of reasoning tools is thus facilitated, for example, a tool for FM edits [TBK09] has been integrated. Nevertheless, current tools do not fully support the composition of FMs or decomposition of an FM into several separated FMs. A conceivable solution would be to integrate our FM operators (insert, aggregate and merge) as additional functionalities inside a mainstream graphical editor. Numerous examples and our case studies indicate that manipulating several FMs requires support for defining and replaying sequences of operations, observing properties as the FMs are manipulated, and organizing all these actions as reusable operations. These observations led us to consider developing a textual, executable language, that can be used in much the same way as scripting languages. Such a scripting language should provide *i)* basic sequencing of FM operations, *ii)* access to FM internals, *iii)* reasoning operations and *iv* composition and decomposition mechanisms. A textual script performs a sequence of operations on FMs. Such operations are *reproducible* and *reusable*. Obtaining the same properties in a graphical editor requires an additional effort, for example, the implementation of an undo/redo system and serialization of the sequence of operations is not straightforward. The scripting aspect of the *FAMILIAR* language is likely to favor *readability* of the specified operations, and *usability* and *productivity* when dealing with decomposition and composition operations on FMs. It should be noted that our use of a textual scripting language does not preclude graphical counterparts built on top of the textual language.
As editors like FeatureIDE and frameworks such as FAMA or SPLOT provide an API, another conceivable solution would be to build an API extension in a mainstream programming language in order to provide support for using composition operators and other FM management operations. While this may be a feasible solution, it would require developers to be knowledgeable about the host language (i.e., Java), many require them to perform repetitive and error-prone actions (e.g., importing an FM or using reasoning operations). If the API was created with simplicity and readability as major objectives, and it is based on a limited set of concepts related to the domain of FMs, then one can argue that the API is an *internal* DSL, written on top of a host language. The external/internal dichotomy is generally used to characterize DSLs. An *external* DSL is a completely separate language and has its own custom syntax. An *internal* DSL is more or less a set of APIs written on top of a host language (e.g., Java). Internal DSL is limited to the syntax and structure of its host language. Both internal and external DSLs have strengths and weaknesses (learning curve, cost of building, programmer familiarity, communication with domain experts, mixing in the host language, strong expressiveness boundary, etc.) [Fow10].

An internal or external DSL should allow an FM user to more quickly build the code they need to manipulate FMs. The facilities provided to the FM users must allow the description of complex operations dedicated to FMs, in both a compact and readable way, while being understandable by an expert who may not necessarily be a software engineer. An external DSL seems particularly adequate in our work as it would provide only the necessary expressive power for anticipated FM

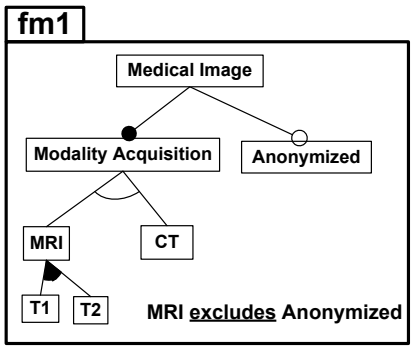
manipulations. In addition, such an external DSL should be used more easily by FM users as the learning curve is expected to be more favorable.
The decision to develop a new DSL (e.g., *when?*, *why?* and *how?*) is a difficult one as it involves both advantages/disadvantages or risks/opportunities [Fow10, vDK98, MHS05]. Essential to this decision is the notion of *domain* that determines the scope of a DSL. Automated analysis of FMs is an active area of research and is gaining importance in the SPL community. Hence the development of an FM manipulation DSL that integrates automated analysis concepts seems particularly adequate. Furthermore, DSLs are considered as "enablers of reuse" [MHS05] and there is a clear opportunity to *reuse operations* already defined in the domain.
We further restrict the DSL to manipulation of *propositional FMs*. Our restriction to propositional FMs means that our DSL does not fully support FMs with feature attributes or constraints expressed in logics that go beyond propositional logic (e.g., [BSRC10, CHE05a, MCHB11]). Moreover, FMs are usually mapped to other artefacts of an SPL (e.g., see [ZJ06, VG07, CA05, HSS+10]). We do not consider the relationship between FMs and other artifacts, i.e., the domain is restricted to FMs.

The predominant idea is that we propose a *textual* language dedicated to the *domain* of FMs. TVL (for Textual Variability Language) [CBH10] has similar characteristics but its focus is on *specifying* FMs and not on manipulating FMs. In particular, the support for reasoning about FMs is not integrated into the language and is provided in the form of a Java library. FeatureIDE [KTS+09], FAMA [FaM08], SPLOT [MBC09] and TVL [CBH10] frameworks do not pursue the objective of managing multiple FMs. Reasoning operations are dedicated to be applied on an unique FM. More importantly and to the best of our knowledge, there is no comprehensive support for composing and decomposing FMs. Moreover, framework users have typically to deal with details about FM imports or reasoning solvers. As a result, manipulations are not encapsulated by operations that hide low-level programming language level details (i.e., additional effort beyond simply manipulating FM concepts is needed) and developers are required to master the framework as well as advanced skills in Java language.

4.2.3 FAMILIAR

The *FAMILIAR* DSL is an executable scripting language that supports manipulating and reasoning about FMs. The next subsections will detail and illustrate the main constructs of the language.
In complement with the abstract example introduced in section 4.1.1, we will use another example taken from one of our case studies, here in the medical imaging domain. Figure 4.5a represents a simplified version of a FM related to the format of a medical image.
FAMILIAR is a typed language that supports both complex and primitive types. Variables representing complex types record a reference to the data whereas other variables record the data value itself[28]. Complex types are *Feature Model*, *Configuration*, *Feature*, *Constraint*, etc. or generic *Set* which represents container values. Primitive types include *String* (e.g., feature names are strings), *Boolean*, *Enum*, *Integer* and *Real*.
Types have accessors for observing the content of a variable. From a feature, one can get its **name**, **parent**, **root**, **children**, etc. Several operators detailed below enable one to manipulate FMs. A classical **if then else** and a loop control structure are also provided.
Statements can then be organized in scripts. *FAMILIAR* provides modularization mechanisms that allow for the creation and use of multiple scripts in a single SPL project, and that support the definition of reusable scripts. The language relies on namespaces to allow disambiguation of variables having

[28]The notions of *reference* and *value* are similar to the ones used in programming languages.



(a) FODA-like representation

$$[[fm1]] = \{$$
$$\{MI, MA, MRI, T1\},$$
$$\{MI, MA, MRI, T2\},$$
$$\{MI, MA, MRI, T1, T2\},$$
$$\{MI, MA, CT\},$$
$$\{MI, MA, CT, AN\},$$
$$\}$$

(b) corresponding set of configurations

Figure 4.5: FM on medical image format.

the same name, and a namespace is associated with each included script. By default, a script makes visible to other scripts all its variables and some **export** and **hide** constructs allows for some fine-tuning if needed. Besides scripts can be parametrized with parameters that can be used as variables.

Importing and Exporting FMs

We provide multiple notations for importing and exporting FMs. A *FAMILIAR* user can *i)* load FMs in these notations (using **FM** constructor) ; *ii)* serialize the FMs in these notations (using **serialize**). Still using the **FM** constructor, *FAMILIAR* also provides a concise notation, notably inspired from FeatureIDE/GUIDSL [KTS+09].

```
1 fm1 = FM (MedicalImage : ModalityAcquisition [Anonymized];//Ano. optional
2                ModalityAcquisition : (MRI|CT); // Xor-group
3                             MRI : (T1|T2)+ ;  // Or-group
4                 MRI excludes Anonymized ; // constraint )
5 sfm1= configs fm1 // set of configurations
6 fm2 = FM ("fm2.tvl") // TVL notation
7 fm3 = FM ("fm3.m") // FeatureIDE/GUIDSL notation
8 serialize fmFoo into SPLOT // export in SPLOT notation
```

In the example of line **1**, the variable *fm1* corresponds to the FM depicted in Figure 4.5a. MedicalImage is the root feature. ModalityAcquisition and Anonymized are child-features of MedicalImage: ModalityAcquisition is mandatory while Anonymized is optional. MRI and CT form a Xor-group and are child-features of ModalityAcquisition. T1 and T2 form an Or-group and are child-features of MRI. " MRI excludes Anonymized" corresponds to a propositional constraint of the FM. The variable *fm1* can then be used, for example in line **5**, to obtain its set of configurations enumerated in Figure 4.5b. FMs can also be imported in other notations using the same **FM** constructor (in line **6** we import a TVL [BCFH10] model while in line **7** we import an FM in FeatureIDE/GUIDSL [KTS+09] notation). Line **8** gives an example of a serialization in SPLOT [MBC09] notation.

Modifications on FMs

The language provides some basic operators for renaming and removing features in FMs. Renaming can be useful when composing or comparing FMs that use different terminology for the same concept (i.e., feature). The following illustrates how features can be renamed:

```
9  oldFeature = parent Analyze // 'Format' feature of mi3
10 newName = strConcat "MI" (name oldFeature)
11 b1 = renameFeature oldFeature as newName // aligning terms
12 assert (b1 eq true) // assert (b1) is equivalent
```

In lines **9-11**, the feature DICOM of FM *mi3* is renamed to MIFormat by concatenating the string *MI* with the old name *Format*. The operator **assert** in line **12** stops the program with an appropriate error message if the renaming is not successful (i.e., *b1* is **false**). A renaming is not successful if the feature to be renamed (e.g., oldFeature in the previous example) does not exist. Similarly, the operator **removeFeature** takes a feature as an argument and removes the feature and its descendants from the FM it belongs to. It also returns **true** or **false** depending whether it is successful or not.

Handling and reasoning about FMs and their configurations

The language also allows *FAMILIAR* users to create FM configurations, and then **select**, **deselect**, or **unselect** a feature. To **select** a feature means that the configuration includes the feature. To **deselect** means that it will not be part of the configuration. To **unselect** means that no decision has been made: the feature is neither selected nor deselected. Each of these configuration manipulation operations returns a boolean value, i.e., true if the feature selected/deselected/unselected does exist.
An example usage of these operations is given below:

```
13 conf1 = configuration mi1 // create a configuration of mi1
14 b1 = select Anonymized in conf1 // feature Anonymized is selected
15 b2 = deselect Anonymized in conf1 // override the previous selection
16 b3 = unselect Anonymized in conf1 // neither selected nor deselected
```

In line **13**, the operator **configuration** creates and initializes a configuration of the FM *mi1*. Lines **14-16** provide examples of the configuration manipulations.
FAMILIAR provides several operators to support reasoning about FMs and configurations. The script below provides examples of the FM manipulation and reasoning operators:

```
17 conf2 = copy conf1
18 nb = counting mi1 // number of valid configurations: 8
19 b1 = isValid conf1
20 b2 = (selectedF conf1) eq (selectedF conf2) // true
21 select Anonymized in conf1
22 confFM = asFM conf1 // convert a configuration into an FM
23 cmp = compare m1 mi2 // refactoring
```

Line **18** computes the number of valid configurations of *mi1*. The **isValid** operator checks whether a configuration is not inconsistent according to its FM (see line **19**). Indeed, a selection or deselection of feature may lead to an invalid configuration, e.g. when two mutually exclusive features are selected. The **isValid** operator can also perform on an FM and determines its *satisfiability* (an FM is unsatisfiable if it represents no configurations). *FAMILIAR* also provides an operator, called **isComplete**, that checks whether a configuration is *complete*, i.e., whether all features have been selected or deselected. The *Configuration* type provides three accessors that return the set of selected, deselected and unselected features: **selectedF**, **deselectedF** and **unselectedF**. Line **20** checks that the set of selected features in both *conf1* and *conf2* are equal, which is true simply because *conf2* is a copy of *conf1*.
Moreover, the operator **asFM** can convert a configuration, say c, into an FM: for each selected feature of c, say f, we add a propositional constraint (i.e., a literal f) to the FM of c and for each deselected

feature of c, say g, we add a propositional constraint (i.e., a negative literal $\bar{g}$) to the FM of c. For example, in line **22**, *confFM* is the FM *mi1 plus* the literal $Anonymized$. **asFM** is typically used when a configuration is not complete.
The **compare** operation is used to determine whether an FM is a refactoring, a generalization, a specialization or an arbitrary edit of another FM. This operation is based on the algorithm and terminology used in [TBK09] (cf. section 4.1.3). Line **23** illustrates comparison capabilities based on FM configuration sets. *cmp* is an *Enum* type whose possible values can be **REFACTORING**, **SPECIALIZATION**, **GENERALIZATION** and **ARBITRARY**. In the example above, *cmp* has the value **REFACTORING** since *mi1* and *mi2* represent the same set of configurations.

Several constructs are dedicated to the management of large number of features and feature models. Dividing FMs into localized and separated parts should be supported, together with the realization of our specific composition and decomposition operators.

Extracting

A first basic mechanism is to "copy" a sub-tree of an FM, including cross-tree constraints involving features of the subtree. It is the role of the **extract** operator, which is purely syntactical and quite limited. It ignores cross-tree constraints that involve features not present in the sub-tree and extracted features must belong to the same sub-tree.

Slicing

The **slice** raise these limitations and implements the operator defined in section 4.1.5. Its syntax is as follow:

```
fmS = slice anFM including | excluding setOfFeatures
```

anFM is the input FM to be sliced and *fmS* is the resulting FM. The set of features that constitutes the slicing criterion can be specified either by inclusion (keyword: **including**) or exclusion (keyword: **excluding**). A specific notation allows an SPL practitioner to select a set of features. Basic operators to perform the union, intersection or difference of feature sets are also provided.

Aggregation

We now describe two important forms of FM composition, aggregate, then merge (insert has a similar syntax). As defined in the previous section, the **aggregate** operator supports cross-tree constraints between features so that separated FMs can be inter-related. The input FMs are aggregated under a synthetic root $synthetic_{ft}$ so that the root features of input FMs are child-mandatory features of $synthetic_{ft}$. In addition, the propositional constraints are added in the resulting FM. For example, the aggregate operator can be used to compose three FMs $fm1$, $fm2$ and $fm3$ together with constraints (see Figure 4.6).

```
1 fm1 = FM (A : B [C] [D] ; D : (E|F) ; C excludes E;) //E&F form Xor-group
2 fm2 = FM (I : J [K] L ; )          // K is optional
3 fm3 = FM (M : (N|O|P)+ ; ) // M, N, O, P form an Or-group
4 cst = constraints (J implies C ; )
5 fm4 = aggregate fm* withMapping cst
6 // last line is equivalent to aggregate { fm1 fm2 fm3 } withMapping cst
7 dfm4 = deads fm4
8 cfm4 = cores fm4
```

All reasoning operations (e.g., **isValid**) can be similarly performed on the new FM resulting from the aggregation. In particular, when FMs are related through constraints, some features may become *dead* or *core* features (cf. section 4.1.3). For example, the feature E is a dead feature in $fm4$ (see Figure 4.6). *FAMILIAR* provides two operators to compute the set of core and dead features of an FM (see line **7–8**).

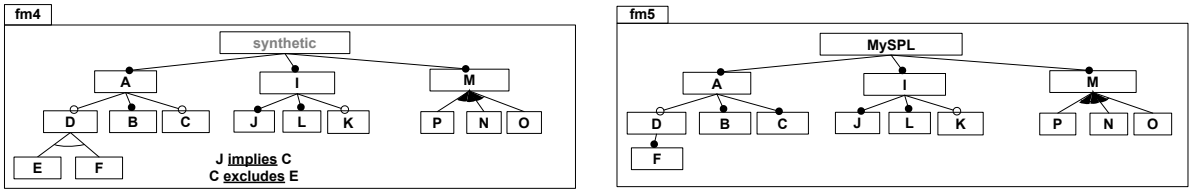


Figure 4.6: Two FMs $fm4$ and $fm5$ at the end of a *FAMILIAR* script execution

Merging

The **merge** operator is dedicated to the composition of FMs that exhibit similar features, i.e. with the same name. In this case, the merge operator can be used to merge the overlapping parts of the FMs and then to obtain an integrated FM. The merge uses name-based matching: two features match if and only if they have the same name.
The syntax the **merge** operator is given in Figure 4.7 where the properties of the merged feature model are summarized with respect to the sets of configurations of input FMs and the mode (intersection, strict union, diff).

Mode	Semantics properties	Mathematical notation	*FAMILIAR* notation
Intersection	$[\![FM_1]\!] \cap [\![FM_2]\!] \cap \ldots \cap [\![FM_n]\!] = [\![FM_r]\!]$	$FM_1 \oplus_\cap FM_2 \oplus_\cap \ldots \oplus_\cap FM_n = FM_r$	fmr = **merge intersection** { fm1 fm2 ... fmn}
Strict Union	$[\![FM_1]\!] \cup [\![FM_2]\!] \cup \ldots \cup [\![FM_n]\!] = [\![FM_r]\!]$	$FM_1 \oplus_\cup FM_2 \oplus_\cup \ldots \oplus_\cup FM_n = FM_r$	fmr = **merge sunion** { fm1 fm2 ... fmn}
Diff	$\{x \in [\![FM_1]\!] \mid x \notin [\![FM_2]\!]\} = [\![FM_r]\!]$	$FM_1 \setminus FM_2 = FM_r$	fmr = **merge diff** { fm1 fm2 }

Figure 4.7: Merge: semantic properties and *FAMILIAR* notation.

Below is part of a script that uses the merge operator in intersection mode:

```
1 mi4 = FM ( MedicalImage: Modality Format Anatomy [Anonymized];
2       Modality: (v10.1|v10) ; Format: (NiftiII|Analyze) ; Anatomy: Brain;)
3 mi5 = FM ( MedicalImage: Modality Format Anatomy [Header];
4       Modality: (v10.1|v10|v9) ; Format: NiftiII ; Anatomy: (Kidney|Brain);)
5 mi_inter = merge intersection { mi4 mi5 }
6 mi_inter_expected = FM ( MedicalImage: Modality Format Anatomy ;
7       Modality: (v10.1|v10) ; Format: NiftiII ; Anatomy: Brain ;)
8 assert (mi_inter eq mi_inter_expected)
```

In line **5**, the merge operator in intersection mode is applied on *mi4* and *mi5* and produces a new FM that can be manipulated through the variable mi_inter. In line **8**, we check that mi_inter is equal to $mi\_inter\_expected$. The binary operator **eq** is specific to variable complex types. In particular, two

variables of FM type are equal if *i)* they represent the same set of configurations, i.e., the **compare** operator applied to the two variables returns **REFACTORING** and *ii)* they have the same hierarchy.

```
9  mi_sunion = merge sunion { mi4 mi5 }
10 n_sunion = counting mi_sunion // number of valid configurations
11 n_expected = counting mi4 + counting mi5 - counting mi_inter
12 assert (n_sunion eq n_expected)
```

In line **9**, the merge operator in strict union mode is applied on *mi4* and *mi5* and produces a new FM that can be manipulated through the variable mi_sunion. In lines **10-12**, we check the following property:

$$[\![mi4]\!] \cup [\![mi5]\!]| = |[\![mi4]\!]| + |[\![mi5]\!]| - |[\![mi4 \cap mi5]\!]| = |[\![mi\_sunion]\!]|$$

using **counting** operations, i.e., the value of n is equal to $[\![mi\_sunion]\!]$.

4.2.4 Environment and Reasoning Back-end

We provide an Eclipse-based development environment for *FAMILIAR* that is composed of i)an Eclipse text editor (including syntax highlighting, formatting, code-completion, etc.), ii) an interpreter that executes the various *FAMILIAR* scripts and iii) an interactive toplevel, connected with graphical editors. *FAMILIAR* is developed in Java language using Xtext [Xte11], a framework for the development of external DSLs. We reuse Xtext facilities to parse *FAMILIAR* scripts and develop the Eclipse text editor. *FAMILIAR* is connected with other languages and framework. Several notations can be used for specifying FMs (SPLOT [MBC09], GUIDSL/FeatureIDE [Bat05, KTS+09], a subset of TVL [BCFH10], etc.). The proposed bridges allow users to import FMs or configurations from their own environments. The connection with the FeatureIDE framework allows for reusing the graphical editors, for both editing and configuring FMs. All graphical edits are synchronized with variables environment and all interactive commands are synchronized with the graphical editors. We also developed a bridge with S2T2[29], a configurator developed at Lero.
All reasoning operations (e.g., **isValid**) on FMs require an efficient representation (i.e., a propositional formula) of the set of configurations. The implementation of the merge and slice operators also relies on propositional formulae. Two reasoning back-ends are internally used in *FAMILIAR* and perform over these propositional formulae: SAT solvers and Binary Decision Diagrams (BDDs). SAT (for satisfiability) solvers aim at deciding the satisfiability problem, i.e., the problem of determining for a given formula whether there is an assignment such that the formula evaluates to true or not. A BDD offers a compact graph-based representation of a Boolean function on a particular ordering of input variables.

The connection with other languages and framework has several benefits. The support of different notations encourages *interoperability* between feature modeling tools. As an FM or a configuration can be exported (using the **save** operation), outputs generated by *FAMILIAR* can be processed by other third party tools, for example, modeling tools when we need to relate FMs to other models [CA05, HSS+10].
Perhaps more importantly, the support of different formats allows one to easily reuse state-of-the-art operations already implemented in existing tools. We reuse an efficient technique to reason about edits described in [TBK09] and implemented in FeatureIDE [KTS+09] to implement the **compare**

[29] http://download.lero.ie/spl/s2t2/

operator. Feature-model-synthesis [30] implements an algorithm to synthesize an FM from a propositional formula (as described in [CW07]). We reuse and adapt some techniques of the algorithm to implement the merge and the slice operators [ACLF11c]. We also reuse some heuristics developed in SPLOT [MBC09] to compile large FMs into BDDs.
A notable benefit for *FAMILIAR* users is that they do not have to deal with implementation details related to solvers. They can directly use operators, thus focusing on domain concepts (FMs, configuration) and avoid accidental complexity. Our practical experiences with *FAMILIAR* gave insights on how to efficiently use solvers within the interpreter. We observed that SAT solver does not scale for **counting** all solutions of FMs whereas BDDs are much faster in counting all solutions and scale better. On the other hand, FMs with a number of features up to 2.000 cannot be compiled to BDD even with the use of heuristics techniques developed in [MWCC08]. As a result, optimal reasoning back-ends (e.g., the choice of BDD for **counting**) are internally selected when an operator of *FAMILIAR* is used and implementable with both back-ends.
For most of the operations, logic encoding prevails over diagrammatic representation: the propositional formula by itself is sufficient to perform reasoning operations. Based on this observation, *FAMILIAR* implements a *lazy strategy*: for all merge, aggregate and slice operations, we only compute the propositional formula of the resulting FM. The hierarchy of the FM is constructed only when needs be, for example, for the purpose of visualization or serialization. The lazy strategy is useful since from our experiments we observed that reconstructing the hierarchy is costly.

4.2.5 Summary

We gave an overview of *FAMILIAR*, a textual and executable domain-specific language that provides a practical support for managing feature models. We describe its main syntactic facilities as well as operators provided so that feature model users can import, export, edit, configure, compose, decompose, configure and reason about feature models. The operators of *FAMILIAR*, combined with modular mechanisms, enable users to separate, compose and reuse feature models. The next section will report on various applications of *FAMILIAR* in different domains, showing some elements of demonstration of its applicability.

4.3 Applications of SoC in Feature Modeling

Again related to Mathieu Acher's PhD Thesis, this section shares material with several papers and summarizes different collaborations with:

- ◇ Robert B. France, Alban Gaignard and Johan Montagnat, in the domain of medical imaging with analysis services and workflow deployed on the grid. Related papers are the SC'10 paper "Managing Variability in Workflow with Feature Model Composition Operators" [ACLF10b] and the Software Quality Journal article "Composing Multiple Variability Artifacts to Assemble Coherent Workflows" [ACG+11].

- ◇ Philippe Lahire, Sabine Moisan and Jean-Paul Rigault, in the domain of video-surveillance systems. It is related to the ICECCS'11 paper "Modeling Variability from Requirements to Runtime" [ACL+11].

[30] https://bitbucket.org/mintcoffee/feature-model-synthesis.

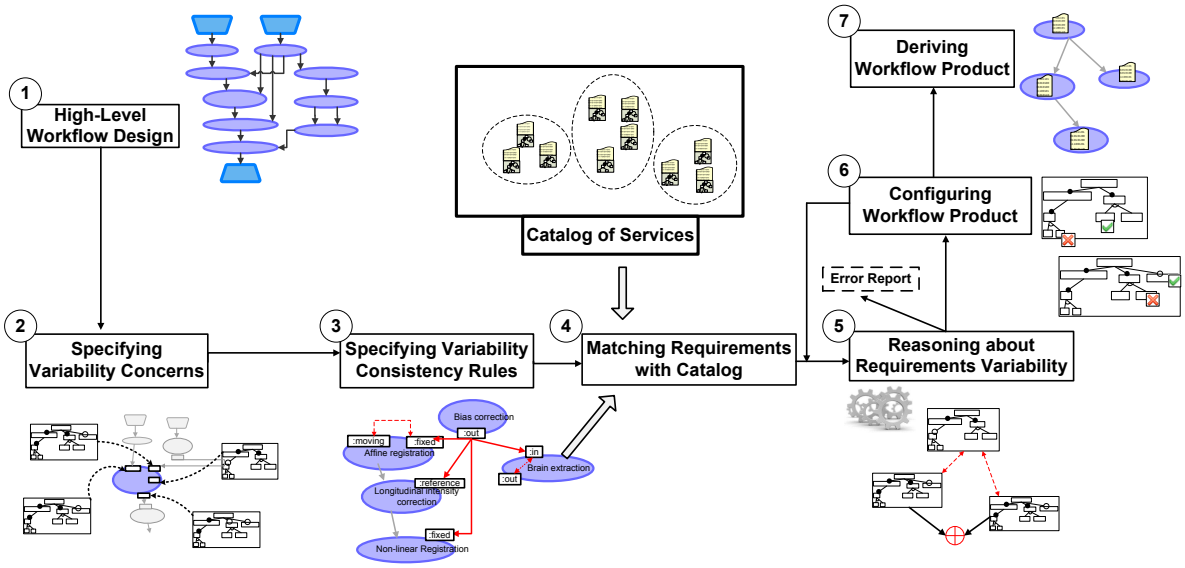


Figure 4.8: Process overview: from design to configuration of the workflow.

◇ Anthony Cleve, Philippe Merle and Laurence Duchien in the domain of reserve engineering feature models related to software architecture. It concerns the ECSA'11 paper "Reverse Engineering Architectural Feature Models" [ACC+11].

We now report on our experience with *FAMILIAR* in four different and representative case studies. We then summarize benefits and limits of both *FAMILIAR* and our operators. For each of the four case studies, we briefly describe the application context, how *FAMILIAR* has been used, the order of complexity in terms of feature modeling and the benefits observed.

4.3.1 Composing Multiple Variability Artifacts to Assemble Coherent Workflows

In some application domain, support for manipulating *multiple SPLs* (i.e., a set of SPLs) may be needed [vO02, vdS04, RW07, HTM09, DGRN10]. When several suppliers compete to deliver similar products, it is interesting to be able to compare and integrate them.
In the medical imaging domain, we proposed a comprehensive modeling process and tooling support (including *FAMILIAR* and a set of domain specific languages) for combining multiple variability artifacts with the purpose of assembling coherent processing chains, called workflows. Separated FMs are used to describe the variability of the different artifacts. At each step of the workflow design, automated reasoning techniques assist medical imaging experts in selecting services from among sets of competing services organized in a catalog while guaranteeing that the composition of services does not violate important constraints.

Process overview. Figure 4.8 gives an overview of the associated multi-step process. Its overall goal is to derive, from an high-level description augmented with variability requirements, a consistent workflow product composed of services offered by the catalog.
In step ① of the process, the workflow designer first develops a high-level description of the workflow that defines the computational steps (e.g., data analyses) that should take place as well as the dependencies between them. The workflow description is then augmented with rich representation

of requirements in order to support discovery, creation and execution of services used to realize the computational steps. In step ②, the workflow designer identifies the variable concerns (e.g., medical image format, algorithm method) for each process of the scientific workflow. The variability of each concern is represented by a FM. Hence several FMs are woven at different, well-located places in each process (e.g., dataport, interface) for specifying the variability of application-specific requirements.
In the general case, some features of a concern may interact with one or more features of other concern(s). In step ③, some application-specific constraints within or across services are typically specified by the workflow designer to forbid some combinations of features. Similarly, some compatibility constraints (e.g., between dataports) can be deduced from the workflow structure and be activated or not by the workflow designer.
In order to ensure that the variability requirements do match the combination of features offered by the catalog, the workflow designer compares, in step ④, the FMs woven in the workflow with the FMs in the catalog of legacy services. In step ⑤, we automatically reason about FMs and constraints specified by the workflow designer in step ① and ②. Constraints propagation and merging techniques are combined to reason about FMs and their compositions. This provides assistance to the workflow designer (detection of errors, automatic selection of features, etc.). To complete the workflow configuration (step ⑥), the designer has to resolve concern FMs where some variability still remains, by performing select/deselect operations. The step ⑥ may be repeated as much as needed in order to allow the designer to proceed incrementally, with step ⑤ also repeated to ensure consistency. In step ⑦, the workflow designer uses the final workflow configuration to identify the services in the catalog that support the combination of features. If more than one service is identified, the workflow designer examines them to chose a best fit or an arbitrary configuration of legacy services. This process has been applied to different real scientific workflows, with up to 25 FMs and two hundreds features [ACG+11].

Variability requirements specification. *FAMILIAR* is used to specify variability requirements (e.g., medical image formats, algorithm method, deployment information, etc.) within services of the workflow. More precisely, *FAMILIAR* is *embedded* into the DSL *Wfamily* which enables skaheolders to:

- ◇ *import* FMs from external files while performing some high-level operations (extraction, renaming/removal of features, etc.). For example, the user can load an existing FM from a catalog, then extract the sub-parts that are of interest and finally specialize the different FMs ;
- ◇ *weave* FMs to specific places of the workflow ;
- ◇ *constrain* FMs within and across services by specifying propositional constraints. Each FM that has been woven has an unique identifier and can be related to another through cross-tree constraints.

Fully-fledged workflow configuration. *FAMILIAR* code is *generated* from the workflow analysis and a *Wfamily* specification, for example, to reason about data compatibility between services. The *FAMILIAR* code is then interpreted to check the consistency of the whole workflow, to report errors to users as well as to automatically propagate choices. Users can incrementally configure, using graphical facilities provided by FeatureIDE editors[31], the various FMs of the workflow. Finally, in order to derive a final workflow product, competing services can be chosen from among sets of services in the catalog using *FAMILIAR* reusable scripts. Our first applications of the process showed

[31] see next section for explanations on bridges between *FAMILIAR* and the FeatureIDE graphical editor.

that the overall approach offers an adequate user assistance and degree of automation for managing the large number of features and FMs, thereby decreasing the effort and time needed.

4.3.2 Modeling Variability From Requirements to Runtime in Video Surveillance Systems

In the development of Video Surveillance (VS) systems, we observed multiple variability factors, both for i) specifying an application, including the environments and contexts where an application is deployed and run (e.g., lighting conditions, information on the objects to recognize), the quality of service required, etc. ii) describing the software platform, including the number of components, their variations due to choices among possible algorithms, the different ways to assemble them, the number of tunable parameters, etc.

Process overview. We applied the principles of Separation of Concerns and represented the variability of the application requirements and the variability of the software platform as two separated FMs [ACL+11]. The software variability is expressed in a dedicated FM, called the *PlatForm Configuration* (*pfc*) model representing a view of implementation modules provided by the software platform. For deriving an actual product, we advocate the use of domain knowledge which contains relevant information to reason about and to select among variants at a higher level of abstraction. The domain variability is expressed through the *Video Surveillance Application Requirement* (*vsar*) model which comprises, in our case, the task specification, the scene context, the object of interests, the Quality of Service (QoS). Confining the variability in a dedicated *space* thus improves the modeling process. During the application engineering process, users can take decisions only related to their know-how and domain.
In Figure 4.9 we present a process that supports rigorous reasoning and user assistance until the application is deployed. In this Figure, we show also the behaviour when adaptations are performed at runtime and the operations required to ensure systematic consistency and end-to-end transformation. The two stakeholders (VS expert and software application engineer) of the approach interact with the FMs during modeling, specialization and transformation. 77 features and 10^8 configurations were present in the *vsar* FM, while 51 features and 10^6 configurations were present in the *pfc* FM. The relationships between the two FMs were described as 39 rules (propositional constraints) relating features across models.

Reachability property checking. Before the execution of a system, FMs are used to verify important properties. Among others, one want to guarantee the *reachability* property, i.e., that for *all* valid specifications and contexts, there exists at least one valid software configuration. In terms of FMs, the reachability property can be formally expressed as follows:

$\forall c \in [\![vsar]\!], c \in [\![slice\, VS_{full}\ including \mathcal{F}_{vsar}]\!]$

where VS_{full} is the aggregation of *vsar*, *pfc* together with cross model constraints, while $\mathcal{F}_{vsar}$ denotes the set of features of *vsar*. Intuitively, if the projection (slice) of the *vsar* features to $[\![VS_{full}]\!]$ is equivalent to the original $[\![vsar]\!]$, the reachability property holds. Otherwise some specifications (i.e., configurations of *vsar*) cannot be reached. The property above can then be implemented with the slice operator and, if needs be, one can also enumerate all products that cannot be realized using the merge operator in diff mode. More precisely, the equation above implies to check if *vsar* is a *refactoring* of $slice\, VS_{full}\ including \mathcal{F}_{vsar}$.

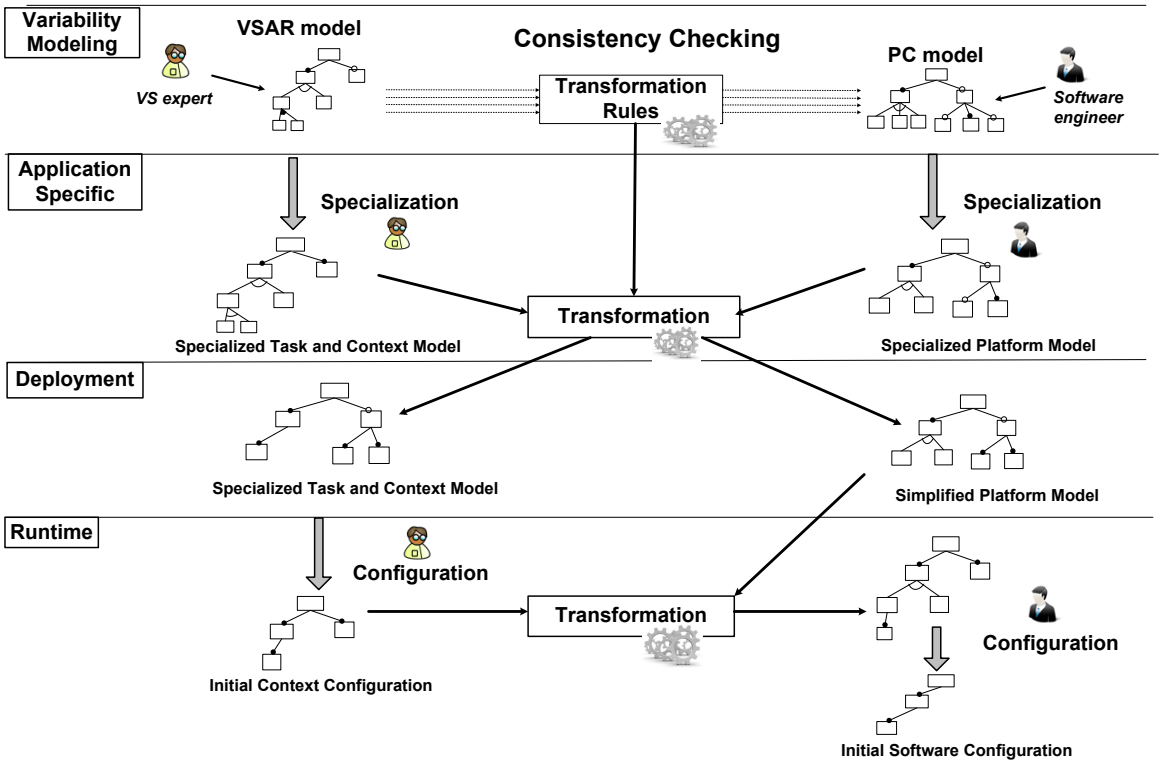


Figure 4.9: From requirements to deployment and runtime: process.

A brute force strategy which consists in enumerating all possible specifications and then checking the existence of a software configuration would be clearly inappropriate, especially in our case where we have more than 10^8 valid specifications and more than 10^6 software configurations. The combined use of **slice**, **compare** and **merge diff** are a much more scalable technique. Without these capabilities, this kind of reasoning would not be made possible for this order of complexity.

Step-wise specialization and choices propagation. In line with specific requirements and deployment scenarios, the video surveillance expert step-wise *specialized vsar* by removing some features, by modifying some feature groups, etc. After the specialization of *vsar*, automated techniques were used to update the software platform FM. To do so, the specialized *vsar* FM and the software platform FM together with rules were aggregated. Finally, we used the **slice** operator on the aggregated FM by only including the set of features related to the software platform.
We observed that from a specification of a context, the possible configurations in the software platform can be highly reduced. We applied the techniques on six different scenarios: the average number of features to consider in the software platform FM was less than 10^4 (instead of 10^6 configurations). In the six scenarios, we *reused i)* the FMs and the constraints and *ii)* automated procedures to control the specialization process and reduce the variability in the software part.

4.3.3 Reverse Engineering Architectural Variability

FAMILIAR was evaluated when applied to FraSCAti [Fra11, MMR+10], a large and highly configurable component and plugin-based system. FraSCAti is an open-source implementation of the Ser-

vice Component Architecture (SCA) standard [sta07], which allows for building hierarchical component architectures with the support of many component and service technologies. As its capabilities grew, FraSCAti has itself been refactored, completely architected with SCA components and become highly configurable in many parts of its own architecture. The goal was to reverse engineer a variability model of the FraSCAti architecture, as the task of manually creating the architectural FM clearly required substantial effort, was daunting, time-consuming and error-prone.

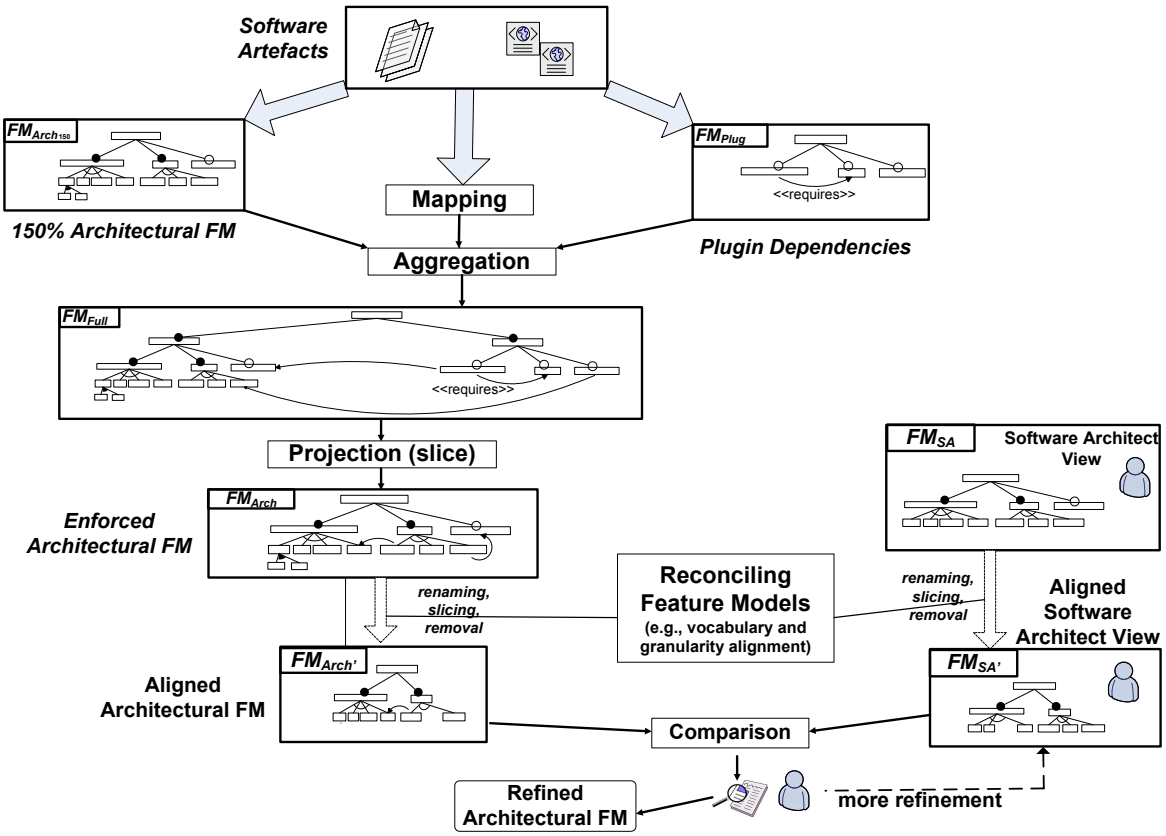


Figure 4.10: Extraction and refinement process of architectural FMs.

Process overview. Figure 4.10 gives an overview of the two main parts of the process, the automatic extraction on the top left part, and the reconciliation and refinement on the bottom part. The starting idea was that the software architecture FM, noted $FM_{Arch_{150}}$, originally produced by an automatic extraction procedure, represents only an over approximation in terms of sets of valid configurations. Hence several sources of information, namely software architecture ($FM_{Arch_{150}}$), plugin dependencies (FM_{Plug}) and the correspondences between software elements and plugins, were *combined* using the **aggregate** operator. Intuitively, the presence of constraints in the aggregated FM reduces the legal combinations of $FM_{Arch_{150}}$'s features. This is the role of the **slice** operator to compute the projection corresponding to the software architecture part of the aggregated FM.
The aggregated FM resulting from the combination of different variability sources and the bidirectional mapping contains 92 features and 158 cross-tree constraints. The slicing technique significantly reduced the over approximation of the original architectural FM (from $\approx 10^{11}$ to $\approx 10^{6}$) so that we obtained a more accurate variability representation of the architecture.

The architectural FM resulting from the automatic extraction was compared with another architectural FM, this time manually designed by Philippe Merle, the FraSCAti software architect. Unfortunately, the direct comparison yielded to unexploitable results, mainly due to the difference of *granularity*, i.e. some features in one FM were not present in the other. Basic manual edits of FMs were unpractical and the **slice** operator was extensively applied as we needed to safely remove features involved in several constraints or that were in the middle of the hierarchy. Once the two FMs have been reconciled, they were compared and differences were identified (e.g., using the **merge** operator in diff mode or the **compare** operator).

4.3.4 Management of Product Line Variability and Software Variability

In SPL engineering, two *concerns* of variability are usually distinguished [PBvdL05, MPH+07]: software variability, related to technical details and hidden from customers (also called internal variability), as opposed to product line (PL) variability, proposing a set of products that are visible to them (also called external variability). In [MPH+07], Metzger et al. proposed a formal and concise approach for *separating* PL variability and software variability and enabling automatic analysis. The two concerns are modeled as two FMs and inter-related by constraints. The authors mention several properties that should be checked when reasoning about the two kinds of variability. As the operators provided in *FAMILIAR* can be combined to support and reason about these two kinds of variability, we revisited several of these properties.
Let fm_{PL} be the FM documenting the PL variability, $fm_{software}$ be the FM documenting the software variability and map_{SoftPL} be the constraints relating the two FMs. The intersection between the set of features of fm_{PL}, denoted $\mathcal{F}_{PL}$, and the set of features of $fm_{software}$, denoted $\mathcal{F}_{software}$, is empty. Furthermore, the mapping between features of fm_{PL} and $fm_{software}$ is not necessarily one-to-one (e.g., a feature in fm_{PL} may imply the selection of three features in $fm_{software}$).

Realized-by property checking. An important property of an SPL is *realizability*, that is, whether the set of products that the PL management decides to offer is fully covered by the set of products that the software platform allows for building. In terms of feature modeling, we want to ensure that for each valid selection/deselection of features of fm_{PL} performed by a customer, there exists at least one corresponding software product described by $fm_{software}$.
To do so we first reason about the relationship between $fm_{software}$ and fm_{PL}. We compute fm_G, the aggregation of fm_{PL} and $fm_{software}$ together with the constraints map_{SoftPL}. In terms of FMs, the realizability property can be formally expressed as follows:

$\forall cp \in [\![fm_{PL}]\!], cp \in [\![slice\ fm_G\ including\ \mathcal{F}_{PL}]\!]$

The *realized-by* property is similar to the *reachability* property described in section 4.3.2: **aggregate**, **slice** and **merge** operators in **diff** and **intersection** modes can be used to automatically check this property. Consequently the *FAMILIAR* scripts developed in the context of video surveillance systems have been reused.

Usefulness property checking. Another important property is to determine whether a product is *useful*, i.e. whether it is a possible realization of a PL member. As argued in [MPH+07], the list of non-useful products is a symptom of unused flexibility of the software platform. Formally, all products are useful if the following relation holds:

$\forall cp \in [\![fm_{software}]\!], cp \in [\![slice\ fm_G\ including\ \mathcal{F}_{software}]\!]$

The usefulness property can be seen as the "symmetric" of the realized-by property, and similar techniques can be applied.

Evolving the FMs. In some circumstances, the realized-by or/and usefulness property may not hold. It can be on purpose, for example, justified by future marketing extensions. It can also be an unexpected property, i.e. an error. *FAMILIAR* scripts can then be used to assist stakeholders in *detecting* the error and *identifying* the missing configurations in fm_{PL} or/and $fm_{software}$. As a result, stakeholders can *correct* fm_{PL}, $fm_{software}$ or the mapping between the two FMs (map_{SoftPL}) and reiterate the process until the expected properties hold. The corrective instructions consist in reusing *FAMILIAR* facilities to edit FMs while checking operations can be performed afterwards. In this case, the *FAMILIAR* language offers a comprehensive solution to manage PL and software variability, including their specifications, their checking and their evolutions.

On scalability. We revisited the approach defended in [MPH+07]. We applied the techniques using the larger example described in [MPH+07]. We successfully retrieved the same results, but our approach is more efficient since we do not enumerate configurations/products as they do. Hence, for larger FMs, the use of our proposed techniques are mandatory since without them the checking of some important properties (e.g., realizability) would not be possible. It is for instance the case for the order of complexity observed in the video surveillance case study.

4.3.5 Summary

Figure 4.11 summarizes the different case studies in which *FAMILIAR* has been involved.
FAMILIAR has been used in different application domains (medical imaging, video surveillance), for different purposes (scientific workflow design, variability modeling from requirements to runtime, reverse engineering FMs) and by different kinds of stakeholders (e.g., domain experts, software engineer). In the different case studies, the main benefits are an adequate and scalable support for Separation of Concerns (SoC) and automated reasoning, the two important requirements that we identified.
In the four case studies, support for SoC was clearly needed, either because FMs were originally separated (variability of services, FMs to be integrated or reconciled) or because the SPL practitioner wanted to modularize the variability (separation of requirements and software variability). Furthermore, the modular mechanisms of *FAMILIAR* allow an FM user to *reuse* FMs and reasoning procedures. In the video-surveillance case study, *vsar* and *pfc* have been reused in the six deployment scenarios as well as parametrized scripts to control the realizability property. In the medical imaging case study, the catalog of FMs has been reused to configure different workflows. Finally, in the case study of section 4.3.4, the PL and software variability can be incrementally managed: edits to the two FMs are applied while reasoning operations can be repeated.
For most of the management activities used in the case studies, a manual intervention is both error-prone, labor-intensive and time-consuming. It is thus very important to be able to automate the decision making process as much as possible. Another important aspect is the ability of techniques to reason about FMs, for example, to infer choices. As previously shown, *FAMILIAR* does provide a set of automated reasoning techniques (operators) that can scale for large FMs and a large number of constraints. The *FAMILIAR* language brings new capabilities to the FM users: without these capabilities, some analysis and reasoning operations would not be made possible, for example, in the video-surveillance case study where an enumerative technique is not adequate.

Case study (domain)	**Stakeholders**	**Complexity**	**Separation of concerns support**	**Automated reasoning support**
Composing Multiple Variability Artifacts (medical imaging, grid computing)	Catalog maintainer, Medical imaging/Grid experts, Workflow designer	hundreds of interrelated features	suppliers' FMs are organized into a catalog ; FMs of the catalog are reused during the design of workflows ; *FAMILIAR* is embedded into another DSL to specify the variability at different places of the workflow.	variability consistency checking of the entire workflow ; choices propagation ; configuration support ;
Modeling Variability From Requirements to Runtime (video surveillance systems)	Video surveillance expert, Software engineer	2 FMs, 77 features and 10^8 configurations in *vsar*, 51 features and 10^6 configurations in *pfc*, 39 cross-tree constraints.	the modeling of requirements and software variability is explicitly separated in two models, *vsar* and *pfc*; the two FMs are reused in six application scenarios ;	consistency checking, specialization checking, reachability property checking, automatic propagation of choices in the software platform, reusable analysis procedures ; variability requirements are incrementally refined ;
Reverse Engineering FMs (component and plugin based systems)	Software architect	the two variability sources contain 92 features, the bidirectional mapping involves 158 cross-tree constraints, while the number of valid configurations varies from $\approx 10^6$ (when reinforced) to $\approx 10^{11}$ (when overapproximated)	*FAMILIAR* code is generated from multiple variability sources in order to construct automatically an FM ;	interactive process and step-wise refinement of the architectural FM by the software architect.
Management of Product Line and Software Variability (any domain)	Products manager, Software engineer	2 FMs, 25 features in fm_{PL} (162 configurations), 11 features in $fm_{software}$ (13 configurations), 13 cross-tree constraints	the two kinds of variability are specified and edited in two separated FMs ;	realized-by and usefulness properties checking ; computation of the differences between the two FMs ; incremental refinement of the two FMs ;

Figure 4.11: *FAMILIAR* and SoC operators: case studies.

CHAPTER 5

Conclusions

This document has described the research we conducted from 2002 to 2011 in the I3S laboratory (Université de Nice Sophia Antipolis - CNRS). This work was made successively in the OCL, RAINBOW and now MODALIS research groups. It concerns the field of software engineering with the general objective of taming the complexity of building and maintaining large software intensive systems. We focused on providing pragmatic integration support for several approaches in the field. The developed solutions always aimed at finding an appropriate trade-off between reliability, so that the manipulated software systems can be used and reused with more confidence, and flexibility, so that the same systems can cope with increasing changes of all forms. In this chapter, we first assess our work following the three research axes that structure it. Section 5.2 then presents a research roadmap for future work.

5.1 Assessment

5.1.1 On Contracting

A first part of our work concerns the provision of contracting techniques and tools for component and service based architectures.

This work starts with the contribution of the *ConFract* system, a contracting system using executable assertions on hierarchical components (cf. section 2.1). We provide a system in which contracts are models at runtime, they are dynamically built from specifications when components are assembled and configured, and updated according to dynamic reconfigurations. Another contribution is in the form of the contracts themselves. They are not restricted to connected interfaces, as object contracts, but new kinds of *composition contracts* are also supported. The contract on external interfaces can be seen as a contract on the usage of the component. Its internal counterpart is more an *"assembly and implementation contract"*. As in object contracts, responsibilities among involved components are automatically determined so that they can be exploited (blame, test oracle, negotiation). One of the main limitations of this work was that only an executable assertion language, `CCL-J`, was used as an input formalism, while other behavioral formalisms could intuitively be interpreted in contracts.

This limitation was raised with the Interact framework. In this framework, the contributions concern the provision of abstractions and automated mechanisms to facilitate software contracting with different kinds of specification formalisms and different component or service based architectures (cf. section 2.2). This framework notably supports the integration of behavioral specification formalisms (Behavior Protocol, TLA) and relies on a central contracting model that handles both compatibility and conformance checking. Some integration of representative formalisms is clearly shown, but a study on what really characterized formalisms that can be *contractualized* would complement and reinforce the obtained results. Another limitation was related to the validation, which only relies on a component-based version. It is raised by the work that follow.

Relying on these advances, we contribute to the ANR FAROS project (cf. section 2.3) by providing both models and platforms related to contracting. This notably shows how the kernel of the Interact

framework was adapted and extended to serve as the central metamodel of the FAROS process. This process drives a model-driven toolchain that supports contracting from the expression of high-level business constraints down to implementation mechanisms. The process also targets different service and component oriented platforms and *ConFract* served as one of the targets, both in the process and for the development of use case applications. In a certain manner, this validates that our contract metamodel is general enough to make management and reasoning possible in such a process and to cope with service orientation. One lesson learned is that a significant amount of time was spent on building the model-driven toolchain dealing with components and services, thus reducing the scope of the validation at the end of the project. Besides it should be noted that this also enables other works to develop other forms of contracts on software architectures [WMD08, WSMD08]. *ConFract*, and its extension in *Interact*, also served as the main example in the Beugnard et al. paper "Contract Aware Components, 10 years after" [BJP10]. This paper revisits the issues discussed in [BJPW99] and presents our contribution as elaborate means of contract management.
Finally, a last contribution in this research axis concerns a testing framework which completely relies on *ConFract* to provide self-testable components with contracts and built-in tests (cf. section 2.4). The framework is only a facilitation structure and coupling with testing techniques to produce tests is an interesting line of research. Recent work on automated testing with contracts [MCLL07] and adaptive random testing [CLOM08] are good candidates. Building the testing framework from the main structuring ideas was really straightforward. We interpret this as a sign of coherent architecture in our contracting system.

5.1.2 On Self-Adaptation

The second part of the presented work is related to the provision of self-adaptive capabilities in and around our contracting systems, i.e. developing concepts and tools around negotiable contracts and providing advanced forms of monitoring systems.
A first contribution is a *ConFract* extension that supports negotiable contracts, in the context of hierarchical components (cf. section 3.1). Inspired by the Contract Net Protocol (CNP) defined in multi-agent systems, it enables configuration and runtime adaptations over components, which are empowered with some capacities of negotiation. Negotiation policies drives the adaptation process by relying on the responsibility model of violated contracts. Two policies, concession-based and effort-based, were proposed. The first allows property relaxations while the second one turns towards the responsible component. A complementary contribution was a model and a supporting run-time infrastructure that allows for reifying non-functional properties in relation with components. It also sports a basic form of compositional reasoning that relates system properties to component properties (cf. section 3.2). These patterns of non-functional properties can be exploited by the negotiation process to propagate effort requests inside the responsible component hierarchy.
We also studied the relation between the obtained negotiable contracts and feedback control loops organized according to the MAPE-K decomposition [IBM01, KC03]. Negotiable contracts are well suited to directly link adaptive behavior to contract violations, and also to provide well-defined verification means when adaptations are triggered. Nevertheless there is no guarantee that the values used during negotiation do not lead to oscillations due to some overshoot in the induced control [HDPT04]. We have partly raised this limitation by providing self-adaptive capabilities over the negotiation mechanisms. We used the very same mechanisms to control some properties such as timeout and to deal with the history of negotiation, detecting some overshoots and deactivating the involved negotiations. But it is also clear that the provided mechanisms are not powerful enough to cover a complete self-healing mechanism. Integrating negotiable contracts into explicit feedback control loops [BDG+09]

would certainly bring several benefits, such as a better integration of adapted control model with static analysis of the control loop behavior [ZC06]. We discuss this line of research in section 5.2.2.
Another contribution concerned monitoring systems, an inescapable piece of any large scale software infrastructure. We proposed a QoI-aware monitoring framework which can deal with multiple clients needing flexible and dynamic QoI needs (cf. section 3.3). The framework allows for instantiating monitoring systems with automatic configuration of all monitoring entities and data sources so that QoI and resource constraints are taken into account. The proposed system relies on a constraint-solving approach to transform QoI needs into probe configuration settings. In the framework, a QoI control component is open to integrate different enforcement algorithms. When resources are unconstrained, monitoring is minimized while guaranteeing all client QoI needs. If they are constrained the QoI enforcement seeks the trade-off between QoI and available resources to maximize utility function provided by clients. This monitoring framework also provides a self-adaptive capability through a control loop that monitors the resource consumption of the other monitoring parts and enforces monitoring constraints. As with the previous contribution, this mainly provides a framework but lacks the integration of sophisticated decision making algorithms. Another limitation concerns the deployment of monitoring systems, which are centralized even they deal with distant data sources and clients. This constitutes interesting lines of research (see below).

5.1.3 On Feature Modeling

The last part of our work concerns feature models, a central formalism to specification and reasoning activities in engineering Software Product Lines (SPLs).
Considering their increasing complexity and the usage of multiple FMs at the same time, we identified the need for supporting *Separation of Concerns* in feature models while enabling reuse of state-of-the-art automated reasoning techniques (cf. section 4.1). Our contribution consisted in a set of composition (aggregate, merge, insert) and decomposition (slice) operators with both a formal semantics definition and an efficient implementation. Their semantics is notably defined in terms of configuration set and hierarchy of the manipulated FMs. The merge operator outputs more compact, and thus more accessible, feature models than current techniques. The slice operator easily generates semantically meaningful decompositions of a feature model. Both operator implementations guarantee that the set of configurations and the hierarchy of the produced feature model are semantically consistent. These operators enable automated reasoning and form a powerful and operational support for SoC applied to feature modeling. As we have grounded their definition on a formal basis, further work could study what is needed to provide a complete support or form an algebra from these operators.
A complementary contribution consisted in the provision of a textual and executable Domain-Specific Language (DSL), named *FAMILIAR* (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning). This language provides a practical support to the previous operators together with additional facilities to import, export, edit, configure and reason about feature models. Resulting scripts can be parametrized and reused through a module mechanism. The language implementation reuses state-of-the-art operations through appropriate connections to other feature modeling frameworks. Internally, SAT and BDD solvers are used to implement some operations. This implementation also comprises a Eclipse-based environment and a stand-alone runtime. Regarding the performance of the two major operators (merge and slice), it has been observed in several case studies that the order of complexity publicly available feature models can be easily handled. However, the operators and language rely on the assumptions that handled feature models are propositional feature models, without any extensions. Some extensions, such as feature attributes, are gaining importance in the SPL field and have to be considered [CBH10, MCHB11]. Moreover, even with its demon-

strated performance, the *FAMILIAR* environment is still a proof of concept. It must be more stabilized and improved so that user experiments and qualitative assessments of the language can be conducted qualitatively. We discuss some perspectives in section 5.2.5.
Different case studies served as validating some aspects of both the operators and the *FAMILIAR* language. *FAMILIAR* was coupled with some other domain specific languages to combine multiple variability artifacts in the medical imaging domain. This constitutes a complete process to build coherent scientific workflows with good user assistance and automation when possible. A second case study demonstrates the usage of multiple FMs to handle variability from requirements to run-time in video-surveillance processing chains. Domain and software variabilities were split into two feature models related by constraints and the configuration spaces was reduced by an order of magnitude in the different studied scenarios. We also used our operators and language to reverse engineering the variability of a component and plugin architectures, i.e. the FraSCAti framework, mixing automated extraction and manual edits. The resulting FM have shown to be correct and useful, as they are now used to drive the new FraSCAti development as a SPL. These case studies demonstrate that our contributions are useful in different application domains and by different people. However, validation could be improved by better defined experiments and more generally, relating feature model composition to the variable software artifacts (model, code) that have also to be composed, is still an open issue. We discus several related lined of research in the next section.

5.2 A Research Roadmap

Our three main research axes all bring interesting lines of research to be developed. We focus here in a research roadmap that naturally seeks to raise limitations of previous works, but which also strives to develop synergies between our domains of expertise. Some starting or ongoing work are discussed along the following paragraphs.

5.2.1 End-to-end Contracting

Our different contributions on contracting software could be put together and extended to provide a more complete solution towards the provision of a reliable and yet flexible approach for component and service based systems. Our contracting systems and frameworks targeted for component-based architectures are likely to be used to express technical contracts inside the architectures, whereas in service oriented architectures, contracts are mainly established with *Service Level Agreements* (SLAs), which focus on properties at an higher level, most often related to business metrics. Both contexts manipulate common concepts of contracting, such as mutual agreement and responsibility.
Naturally, a more general approach would enable to define high-level SLA, then refines them into contracts related to implementing components and subcomponents. Some recent work show the possibility of expressing different types of contract on different *points* of a service and component architecture [WMD08, WSMD08]. Relating different forms of contracts is a line of research that would necessitate to integrate negotiation mechanisms similar as ours, so that general functioning *modes* can be determined to organize self-adaptation at a coarse grain, as well to use QoI-aware monitoring systems with an overall control over its resource consumption. With different forms of contracts, deployed at different locations and with different moments at which they are relevant [BJP10], different input formalisms will have to be taken into account, and salient combination of different forms of specification and their verification techniques will have to be studied.

5.2.2 Model-Driven Construction of Self-Adaptive Systems

Through our work on negotiable contracts and on adaptive monitoring, we face some major issues in engineering self-adaptive systems. Despite some adapted mechanisms to organize the self-adaptive behavior, there were possibilities of incorrect behavior, either because the control model based on thresholds was too weak to ensure relevant properties of control [HDPT04], or because this behavior was not represented as a statically analyzable model [ZC06].
An interesting line of research is then to provide models, techniques and tools to facilitate the engineering of self-adaptive systems in large software systems. Many challenges have to be tackled in this domain [CLG+09], such as making control loops explicit [BDG+09], providing appropriate architectures fostering reuse of loop and loop elements, supporting these efficiently at runtime while having strong verification and validation capabilities.

The SALTY project. From the end of 2009, I am leading a three-years ANR funded project, SALTY[32] (for Self-Adaptive very Large disTributed sYstems). Its aim is to provide a software framework to implement self-adaptive systems using model-driven engineering techniques to abstract away technological contingencies while relying on software architecture standards, such as SCA [Ope07], at runtime. Developers and maintainers of large scale infrastructures should then be able to design control loops with explicit self-adaptive capabilities at runtime, while monitoring a very large number of entities and events and coordinating feedbacks of different nature and objectives. Several case studies are used in the project. A first one concerns the regulation in load and frequency of data gathered from the geo-tracking of very large truck fleet (10.000 to 100.000 trucks) [MAC+10]. Another is related to the generic self-regulation of grid and high-throughput computing middlewares [CKM+10]. As many other studies on the state of the art have shown [CLG+09, ST09], in this project, we acknowledge the fact that the domain is rich of frameworks providing support for building self-adaptive systems [KPGV03b, GCH+04, LPH04b], for example by following the MAPE-K autonomic decomposition [KC03], but also that the resulting feedback control loops are not explicit enough. To solve the problem of the opacity and the non-scalability of such approaches, some recent research works suggest that, "*the feedback loops that control self adaptation must become first-class entities*" [BDG+09]. Despite recent works with more advanced frameworks [SBD08, HLM+09, RRS+09] there is still major issues in coordination of several control loops and large scale management.
Facing these issues, the SALTY framework should provide methodological processes and architectural models to make explicit loops as assemblies of reusable loop elements. Regarding decision making, the framework strives to explore automatic computation of policies at run time using learning techniques (reinforcement learning and neuro-fuzzy approaches) while enforcing the SASO properties from control engineering (*stability*, *accuracy*, *settling time*, *overshoot* [HDPT04]). Some large scale coordination algorithms are also under development by some partners in the project. All together, these elements should be transformed into a platform dependent representation, based on the SCA standard. At runtime, it will reuse an implementation of feedback control loops architected with SCA components, which will be an evolution of the current SPACES system [NRS10].

Ongoing work. In this context, the ongoing PhD Thesis of Filip Krikava aimed at providing an abstract model to represent loop and loop elements. A first architectural model has already been designed and prototyped [KCBF11, KC11]. In the proposed architecture, each part of the feedback

[32] https://salty.unice.fr

control loop is uniformly and explicitly designed as a first-class adaptive element. Making these elements explicit allows the architect to reason about system modeling, while capturing different patterns of interactions and controls (coordination of loops, adaptive monitoring, loops over controllers, etc.). Code generation from the architecture avoids painful details of low-level system implementation. Small experiments on the Condor middleware[33] have shown that some non-trivial control can be applied while checking at runtime that the loops are stable and robust. Ongoing work consists in evolving the architectural model to support composite loop elements and to represent deployment of large number of loops. At longer term, we expect that reusable patterns of loop could be modeled with our solution. A supporting DSL is going to be provided so to ease both description and reuse of these elements. This is obviously related to issues of model at runtime, as we need to find a trade-off between scalability and explicit representation. Prototyping and experimentation are also conducted in parallel, so that the resulting architectures can be put to the test. Similar deployments on two different middlewares, GLite and Condor, should bring interesting results in terms of resulting adaptive behavior and reuse of the control loops and of their elements.

5.2.3 Checking and Contracting Self-Adaptive Systems

With the explicit architectural model evoked above, another objective is to integrate verification techniques at appropriate times for ensuring consistency and behavior of the control loops. Based on petri nets, there have been some work with significant impact bringing some possible solutions to the static analysis and code generation of verified self-adaptive behavior [ZC06]. More recently, an approach based on a DSL was proposed for sense/compute/control applications [CBCL11]. It notably relies on model-checking of the built control loops while generating code skeletons in such a way that it provides some guarantees on the usage of loop elements. This approach also uses a form of *interaction contracts* to express some coordination rules at some points of the control loops.
In order to provide similar mechanisms on larger and more complex control loops, we envisage to be able to build a behavioral model of several coordinated loops from our architectural model, so that it can be model-checked. Moreover we would like to provide more versatile contracts than the interaction contracts of [CBCL11]. Using assertions together with some extensions should provide specifications that can be statically checked for compatibility among loop elements, while being also used when testing the loop assembly. Besides, these new forms of contracts could also referred to some non-functional properties, such as the quality of information in all the data flow of a given loop.

5.2.4 Software Product Lines of Self-Adaptive Systems

In our contributions on feature models, we handled variability from requirements to runtime, notably by modeling both the context and the software variants with feature models [ACF+09]. This is related to the field of dynamic SPLs, i.e. SPL in which there is some variability of the context (information accessible at runtime) and dynamic binding of the system to a specific variant [CHS+08]. There have been several interesting work that use aspect-orientation with some variability handling techniques, to unify design-time and runtime variabilities [PBCD11] or to reduce the combinatorial explosion of variants when handling adaptations of software modes according to the context [MBJ+09].
A first interesting line of research is to integrate this dynamic SPL principles with our ongoing work of framework for self-adaptive systems. A more original line is to consider the usage of SPL engineering techniques for the construction of the loop architectures. We already face problems of composition and reuse among our loops and loop elements. Capturing typical loop interactions in variable patterns

[33] http://www.cs.wisc.edu/condor/

with some consistent configuration looks a promising way to us. This could also be related to work on testing SPL [PSK+10], so that the process of testing feedback control loops could also benefit from these results.

5.2.5 Scalable Feature Modeling Composition

This naturally leads to lines of research directly related to our contribution on feature modeling composition. As analyzed in the previous sections, some improvements are necessary to stabilize and improve the *FAMILIAR* DSL and its underlying implementation. This should allow for using *FAMILIAR* for larger and better defined experiments. As *FAMILIAR* is now used to manage its variability, we notably envisage an empirical study on the FraSCAti SPL to trace evolution among features of successive versions. We expect some interesting outputs on some more long term usage of *FAMILIAR*, so that its adequacy can be analyzed.
Regarding the language and the composition operators, the complete integration of feature attributes, like in TVL [CBH10], is one of the priorities. As for the implementation of the merge and the slice operators, the current version is limited to an implementation based on BDDs. An ongoing work develop efficient techniques to encode propositional formulae fed to SAT solvers, adapting recent results from [SLB+11]. We plan to make a systematic comparison of BDD and SAT based implementations, and use it to offer an optimal support into *FAMILIAR*.
Besides, this support will encompass some new functionalities. When merging or comparing feature models in an open domain, suppliers may use different hierarchies, concepts and vocabulary, thus implying some *alignment* activities to relate concepts. In our applications, we mainly rely on a common basis, e.g., through a single ontology [FJFA+10], or we use *FAMILIAR* scripts to restructure a hierarchy with renaming and removing actions. A possible extension is to consider a semi-automatic process for identifying features that are equivalent, intuitively by reusing and adapting matching techniques [ES07]. Coupled with these alignment techniques, an interesting issue is the provision of a better *diff* operator that should combine our *merge diff* operator working on configuration sets with some syntactic comparison of the hierarchies.

5.2.6 On the Relation of Features to Other Models

Following an approach that extensively uses feature models in SPL engineering, a very important problem that we did not tackle so far is the relationship between feature models and other models in the broad sense of term, i.e. textual requirements, UML diagrams and models, source code.
A first issue is related to the extraction of feature models to abstract the variability of other models. Our contribution in extracting architectural feature models from the FraSCAti paves the way for investigating further the combination of multiple information sources [CKK06, WCR09] with some expert knowledge. To do so, we will need to face several challenges, in automating the extraction while putting the expert in the process, providing a result of quality in terms of features and hierarchies, and finding scalable implementation techniques.
A second issue concerns the realization of variability in other models. Currently, two different approaches are used to implement variability in SPLs at the code or model levels, by relying either on annotative (aka model pruning) or compositional (aka model merging) techniques [PKGJ08a, Käs10, HKW08, SZLT+10, PVL+10, PCBD10]. Numerous contributions have been made to cope with realization at the code level, but we envisage to focus more on variability in models. Most existing work focus on checking the syntactical consistency of model products and finding salient semantic properties that can be expressed and checked during pruning or merging models. In the composi-

tional approach, Saval et al. notably identified three challenges when merging models [SPHM09], co-evolution of model fragments, lack of variability in the base model and scoping of model fragments. There is thus a need to provide techniques and tools to model and relate appropriate forms of variability. Facing these challenges will also enable us to investigate how feature models can participate in a compositional approach involving a large number of inter-related models, i.e. compositional SPL [Bos10].
Moreover an ANR project named YourCast, starting in 2012 in our research group, aims at providing an SPL for information broadcast systems. *FAMILIAR* will be extensively throughout the SPL, but new solutions will also have to be developed, enabling us to study some of the evoked research lines.

Bibliography

[ACC09] Zied Abid, Sophie Chabridon, and Denis Conan. A framework for quality of context management. In *QuaCon*, pages 120–131, 2009. 81, 89

[ACC+11] Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Reverse Engineering Architectural Feature Models. In *5th European Conference on Software Architecture(ECSA'11), long paper*, LNCS, page 16, Essen (Germany), September 2011. Springer. 8, 110

[ACD+05] Alain Andrieux, Karl Czajkowski, Asit Dan, Kate Keahey, Heiko Ludwig, Toshiyuki Nakata, Jim Pruyne, John Rofrano, Steve Tuecke, and Ming Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG, September 2005. 36

[ACF+09] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Context and Dynamic Adaptations with Feature Models. In *4th International Workshop Models@run.time at Models 2009(MRT'09), workshop*, , pages 89–98, Denver, Colorado, USA, October 2009. CEUR-WS.org. 124

[ACG+11] Mathieu Acher, Philippe Collet, Alban Gaignard, Philippe Lahire, Johan Montagnat, and Robert France. Composing Multiple Variability Artifacts to Assemble Coherent Workflows. *Software Quality Journal Special issue on Quality Engineering for Software Product Lines - Accepted with minor revisions*, December 2011. 8, 101, 109, 111

[Ach08] Mathieu Acher. Vers une ligne de services pour la grille: application à l?imagerie médicale. Master's thesis, Université de Nice Sophia-Antipolis, Sophia Antipolis, France, June 2008. 8

[Ach11] Mathieu Acher. *Managing Multiple Feature Models: Foundations, Language and Applications*. PhD thesis, University of Nice Sophia Antipolis, Nice, France, September 2011. 8, 95, 99

[ACL+11] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Variability from Requirements to Runtime. In *16th International Conference on Engineering of Complex Computer Systems(ICECCS'11)*, , Las Vegas, April 2011. IEEE. 94, 101, 109, 112

[ACLF09] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Composing Feature Models. In *2nd International Conference on Software Language Engineering(SLE'09), long paper*, LNCS, page 20, Denver, Colorado, USA, October 2009. LNCS. 8, 92

[ACLF10a] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Comparing Approaches to Implement Feature Model Composition. In *6th European Conference on Modelling Foundations and Applications(ECMFA)*, volume 6136 of *LNCS*, pages 3–19, France, June 2010. Springer. 8, 100

[ACLF10b] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Managing Variability in Workflow with Feature Model Composition Operators. In *9th International Conference on Software Composition(SC'10)*, volume LNCS of *Software Composition*, page 16, Malaga, Spain, June 2010. Springer. 8, 94, 109

[ACLF11a] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing(SAC)*, , Taiwan, March 2011. Programming Languages Track, ACM. 101

[ACLF11b] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support. In *Fifth International Workshop on Variability Modelling of Software-intensive Systems(VaMoS'11)*, VaMoS, Namur, Belgium, January 2011. ACM. 101

[ACLF11c] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Slicing Feature Models. In *26th IEEE/ACM International Conference On Automated Software Engineering(ASE'11), short paper*, , Lawrence, USA, November 2011. IEEE/ACM. 8, 92, 99, 109

[ACMS06] S. Agarwala, Yuan Chen, D. Milojicic, and K. Schwan. Qmon: Qos- and utility-aware monitoring in enterprise systems. In *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, June 2006. 82

[Aeg01] J.O. Aegedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University Of Oslo, 2001. 22

[AG97] R. J. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Soft. Eng. and Methodology*, 6, July 1997. 20

[AG02] Colin Atkinson and Hans-Gerhard Groß. Built-in contract testing in model-driven, component-based developmet. In *ICSR7 2002 Workshop on Component-based Software Development Processes*, Austin, Texas, apr 2002. 44

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proc. of GPCE'2006*, pages 201–210. ACM, 2006. 100

[AHAES08] M. Anwar Hossain, P.K. Atrey, and A. El Saddik. Context-aware qoi computation in multi-sensor systems. In *Mobile Ad Hoc and Sensor Systems, 2008. MASS 2008. 5th IEEE International Conference on*, pages 736–741, Oct 2008. 89

[AJTK09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model superimposition in software product lines. In Richard F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 4–19. Springer, 2009. 6

[AK09] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology (JOT)*, 8(5):49–84, July/August 2009. 91, 92, 93

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993. 25, 32, 33, 35, 37

[BAB+00] Gordon S. Blair, Anders Andersen, Lynne Blair, Geoff Coulson, and D. Sanchez. Supporting dynamic qos management functions in a reflective middleware platform. *IEE Proceedings - Software*, 147(1):13–21, 2000. 55, 79

[BAM03] Jean Michel Bruel, João Araújo, and Ana Moreira. Using aspects to develop built-in tests for components. In *4th AOSD Modeling With UML Workshop, Workshop of the UML 2003 Conference*, oct 2003. 44

[Bat05] Don S. Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *LNCS*, pages 7–20, 2005. 108

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May 2000. Volume 2. 2, 10, 11, 66

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42:22–27, October 2009. 54

[BCFH10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *VaMoS'10*, pages 159–162, 2010. 6, 104, 108

[BCL+04] Eric Bruneton, Thierry Coupaye, Mathieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In *ICSE 2004 - CBSE7*, volume 3054 of *LNCS*. Springer Verlag, May 2004. 10, 13, 18, 21, 26, 66

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience,special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12), September 2006. 10, 21, 66, 86

[BCW11] Kacper Bąk, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin / Heidelberg, 2011. 6

[BDG+09] Yuriy Brun, G. Di Marzo Serugendo, Cristina Gacek, Holger Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009. 4, 5, 64, 87, 120, 123

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 350, Washington, DC, USA, 2004. IEEE Computer Society. 89

[BG99] Christian Becker and Kurt Geihs. Generic QoS specification for CORBA. In *Kommunikation in Verteilten Systemen (KiVS'99)*, Darmstadt, Germany, Mars 2-5 1999. 65

[BG05] Luciano Baresi and Sam Guinea. Towards dynamic monitoring of ws-bpel processes. In *ICSOC 2005, Third International Conference of Service-Oriented Computing, volume 3826 of Lecture Notes in Computer Science*, pages 269–282. Springer, 2005. 36

[BGP06] L. Baresi, S. Guinea, and P. Plebani. WS-Policy for service monitoring. In *Technologies for E-Services, 6th International Workshop, TES 2005, Trondheim, Norway, September 2-3, 2005, Revised Selected Papers*, volume 3811 of *LNCS*, pages 72–83. Springer, 2006. 80

[Bin96] Robert V. Binder. Testing object-oriented software : A survey. *Journal of Sofware Testing, Verification and Reliability*, 6(125-252), 1996. 44

[BJP10] Antoine Beugnard, Jean-Marc Jézéquel, and Noël Plouzeau. Contract aware components, 10 years after. In Javier Cámara, Carlos Canal, and Gwen Salaün, editors, *WCSI*, volume 37 of *EPTCS*, pages 1–11, 2010. 120, 122

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32:38–45, July 1999. 3, 44, 120

[BKLW95] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality Attributes. Technical Report CMU/SEI-95-TR-021, CMU/SEI, December 1995. 79

[BKS03] T. Buchholz, A. Kupper, and M. Schiffers. Quality of context information: What it is and why we need it. In *Proceeding of the 10th HP-OVUA Workshop, Geneva, Switzerland*, 2003. 80, 89

[BLH00] Z. Balogh, M. Laclavik, and L. Hluchy. Model of Negotiation and Decision Support for Goods and Services. In *Proceedings of XXIInd International Colloquium ASIS 2000 - Advanced Simulation of Systems*, Ostrava, Czech Republic, 2000. 65

[BMFGI06] Sonia Ben Mokhtar, Damien Fournier, Nikolaos Georgantas, and Valérie Issarny. Context-Aware Service Composition in Pervasive Computing Environments. In *Rapid Integration of Software Engineering Techniques, Second International Workshop : RISE 2005*, pages 129–144, Heraklion, Crete, Grèce, 2006. 36

[Bos10] Jan Bosch. Toward compositional software product lines. *IEEE Software*, 27:29–34, 2010. 126

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *DAC'90*, pages 40–45. ACM, 1990. 99

[BRL07] Fabien Baligand, Nicolas Rivierre, and Thomas Ledoux. A declarative approach for qos-aware web service compositions. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 422–428. Springer, 2007. 38

[BRO+02] G. Brahnmath, R. Raje, A. Olson, B. Bryant, M. Auguston, and C. Burt. A Quality of Service Catalog for Software Components. In *SESEC'2002*, 2002. 79

[BS03] Mike Barnett and Wolfram Schulte. Runtime Verification of .NET Contracts. *Journal of Systems and Software*, 65(3):199–208, 2003. 11, 12, 13, 20, 21

[BSRC10] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*, 2010. 92, 93, 94, 95, 98, 101, 103

[BTPT06] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Web Services, 2006. ICWS '06. International Conference on*, pages 63–71, Sept. 2006. 80

[BV02] M. Bertoa and A. Vallecillo. Quality Attributes for COTS Components. In *ECOOP'2002 QAOOSE Workshop*. Springer LNCS, 2002. 79

[CA05] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, volume 3676 of *LNCS*, pages 422–437, 2005. 6, 103, 108

[CBCL11] Damien Cassou, Emilie Balland, Charles Consel, and Julia L. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 431–440. ACM, 2011. 124

[CBH10] Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution*, 2010. 103, 121, 125

[CC05] Hervé Chang and Philippe Collet. Fine-grained Contract Negotiation for Hierarchical Software Components. In *31th EUROMICRO-SEAA Conference 2005 - CBSE Track*, , pages 28–35, Porto, Portugal, August 2005. IEEE Computer Society Press. 8, 10, 54

[CC06] Hervé Chang and Philippe Collet. Négociation de contrats : des systèmes multi-agents aux composants logiciels. *RSTI - Série L'Objet (RSTI-Objet)*, 12(4):73–102, December 2006. 8

[CC07a] Hervé Chang and Philippe Collet. Compositional Patterns of Non-Functional Properties for Contract Negotiation. *Journal of Software (JSW)*, 2(2):52–63, August 2007. 8

[CC07b] Hervé Chang and Philippe Collet. Patterns for Integrating and Exploiting Some Non-Functional Properties in Hierarchical Software Components. In *14th IEEE International Conference on the Engineering of Computer-Based Systems(ECBS'07)*, , pages 83–92, Tucson Arizona, USA, March 2007. IEEE Computer Society Press. 8

[CCOR06] Hervé Chang, Philippe Collet, Alain Ozanne, and Nicolas Rivierre. From Components to Autonomic Elements using Negotiable Contracts. In *3rd International Conference on Autonomic and Trusted Computing(ATC'06)*, volume 4158 of *LNCS*, pages 78–89, Wuhan, China, September 2006. Springer Verlag. 8, 54

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 5

[Cha04] Hervé Chang. Mécanismes de négociation pour composants logiciels contractualisés. Master's thesis, Laboratoire I3S - Université de Nice-Sophia Antipolis, June 2004. 8

[Cha07] Hervé Chang. *Négociation de contrats dans les systèmes à composants logiciels hiérarchiques*. PhD thesis, Université de Nice - Sophia Antipolis, Sophia Antipolis, France, December 2007. 8

[CHE05a] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*, pages 7–29, 2005. 103

[CHE05b] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. 93

[CHS+05] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the java platform. *Softw., Pract. Exper.*, 35(2):123–157, 2005. 79

[CHS+08] Andreas Classen, Arnaud Hubaux, Franciscus Sanen, Eddy Truyen, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, Patrick Heymans, and Wouter Joosen. Modelling variability in self-adaptive systems: towards a research agenda. In *Proceedings of International Workshop on Modularization, Composition and Generative Techniques for Product-line Engineering,*, pages 19–26, October 2008. 124

[CKK06] Krzysztof Czarnecki, Chang Hwan Peter Kim, and Karl Trygve Kalleberg. Feature models are views on ontologies. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 41–51, Washington, DC, USA, 2006. IEEE Computer Society. 125

[CKM+10] Philippe Collet, Filip Křikava, Johan Montagnat, Mireille Blay-Fornarino, and David Manset. Issues and Scenarios for Self-Managing Grid Middleware. In *Workshop on Grids Meet Autonomic Computing, in association with ICAC'2010(GMAC 2010), workshop*, , page 10, Washington, DC, USA, June 2010. ACM. 8, 123

[CL02] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, 2002. 2, 10

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. pages 1–26, 2009. 2, 4, 123

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 71–80. ACM, 2008. 120

[CLP05] Ivica Crnkovic, Magnus Larsson, and Otto Preiss. Concerning predictability in dependable component-based systems: Classification of quality attributes. In Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems III*, volume 3549 of *Lecture Notes in Computer Science*, pages 257–278. Springer Berlin / Heidelberg, 2005. 10.1007/11556169_12. 79

[CM04] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In Liang-Jie Zhang and Mario Jeckle, editors, *Web Services*, volume 3250 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30209-4_13. 36

[CMOR07] Philippe Collet, Jacques Malenfant, Alain Ozanne, and Nicolas Rivierre. Composite Contract Enforcement in Hierarchical Component Systems. In *6th International Symposium on Software Composition(SC'07)*, volume 4829 of *LNCS*, pages 18–33, Braga, Portugal, March 2007. Springer Verlag. 7, 21

[CN01] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001. 3, 91

[COR06] Philippe Collet, Alain Ozanne, and Nicolas Rivierre. Enforcing Different Contracts in Hierarchical Component-Based Systems. In *5th International Symposium on Software Composition(SC'06)*, volume 4089 of *LNCS*, pages 50–65, Vienna, Austria, March 2006. Springer Verlag. 7, 21, 35

[COR07] Philippe Collet, Alain Ozanne, and Nicolas Rivierre. Towards a Versatile Contract Model to Organize Behavioral Specifications. In *33rd International Conference on Current Trends in Theory and Practice of Computer Science(SOFSEM'07)*, volume 4362 of *LNCS*, pages 844–855, Harrachov, Czech Republic, January 2007. Springer Verlag. 7, 21

[CRCR05] Philippe Collet, Roger Rousseau, Thierry Coupaye, and Nicolas Rivierre. A Contracting System for Hierarchical Components. In *Component-Based Software Engineering, 8th International Symposium(CBSE'2005)*, volume 3489 of *LNCS*, pages 187–202, St-Louis (Missouri), USA, May 2005. Springer Verlag. 7, 10

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable processing of context information with cosmos. In *DAIS*, pages 210–224, 2007. 86, 87, 89

[CW07] K. Czarnecki and A. Wąsowski. Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34, 2007. 92, 93, 99, 109

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In *Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001. 11, 21

[DBFC+08] Laurence Duchien, Mireille Blay-Fornarino, Philippe Collet, Nicolas Rivierre, Vincent Hourdin, Sébastien Mosser, Stéphane Lavirotte, Lionel Seinturier, and Jean-Yves Tigli. Métamodèles de plates-formes. Technical Report F-2.3, RNTL FAROS, July 2008. xi, 38, 39, 41, 42, 43

[DC01] Jim Dowling and Vinny Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag. 55

[DC06] Daniel Deveaux and Philippe Collet. Specification of a Contract Based Built-In Test Framework for Fractal. In *Fractal CBSE Workshop at ECOOP'06*, , page 4, Nantes, France, July 2006. ObjectWeb. 7, 44

[DFJ01] Daniel Deveaux, Patrice Frison, and Jean-Marc Jézéquel. Increase Software Trustability with Self-Testable Classes in Java. In D. D. Grant and L. Sterling, editors, *Proceedings 2001 Australian Sofware Engineering Conference*, pages 3–11, Canberra - Australia, August 2001. ASWEC'2001, IEEE Computer Society. 44, 47, 48

[DGRN10] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *Journal of Systems and Software*, 83(7):1108–1122, 2010. 93, 110

[DJP04] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Extra-Functional Contract Support in Components. In Ivica Crnkovic, Judith Stafford, Heinz Schmidt, and Kurt Wallnau, editors, *ICSE 2004 - CBSE7*, volume 3054 of *LNCS*, pages 217–232. Springer Verlag, May 2004. 20

[DL03] P-C. David and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. In *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, volume 2893 of *Lecture Notes in Computer Science*, pages 1–14, Paris, France, November 2003. Springer-Verlag. 55

[DLLC09] Pierre-Charles David, Thomas Ledoux, Marc Léger, and Thierry Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009. 49

[Ecl10] Eclipse.org. Eclipse modeling framework, July 2010. 86

[EFHC02] E. Eskenazi, A. Fioukov, D. Hammer, and M. Chaudron. Estimation of Static Memory Consumption for Systems Built from Source Code Components. In *EUROMICRO-SEAA'2002*. IEEE Computer, 2002. 79

[ES07] Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2007. 125

[FaM08] FaMa. http://www.isa.us.es/fama/, 2008. 103

[FF01] Robert B. Findler and Matthias Felleisen. Contract Soundness for Object-Oriented Languages. In *Proceedings of OOPSLA'2001*, 2001. 18

[FJFA+10] Martin Fagereng Johansen, Franck Fleurey, Mathieu Acher, Philippe Collet, and Philippe Lahire. Exploring the Synergies Between Feature Models and Ontologies. In *International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010)(SPLC'10 (Volume 2))*, volume 2 of *SPLC'10 (Volume 2)*, pages 163–171, Jeju Island, South Korea, September 2010. Lancester University. 125

[FK98a] Svend Frölund and Jari Koistinen. QML: a language for Quality of Service Specification. Technical Report HPL-98-10, Software Technology Laboratory, Hewlett-Packerd, February 1998. 65

[FK98b] Svend Frölund and Jari Koistinen. Quality of Service in Distributed Object Systems Design. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe (New Mexico)*. USENIX, April 27-30 1998. 20

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings American Mathematical Society Symposium in Applied Mathematics*, volume 19, pages 19–31, 1967. 3

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 2010. 102, 103

[FR07] Robert B. France and Bernhard Rumpe. Does model driven engineering tame complexity? *Software and System Modeling*, 6(1):1–2, 2007. 43

[Fra11] FraSCAti. https://wiki.ow2.org/frascati/Wiki.jsp?page=Main, 2011. 113

[FSJB99] P. Faratin, C. Sierra, N.R. Jennings, and P. Buckle. Designing Flexible Automated Negotiators: Concessions, Trade-Offs and Issue Changes. Technical Report RP-99-03, Institut d'Investigacio en Intel.ligencia Artificial Technical Report, 1999. 65

[GB99] E. Gamma and K. Beck. Junit: A cook's tour. *Java Report*, pages 27–38, 1999. 45

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkist. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, October 2004. 123

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994. 1, 86

[GMB06] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *Proceedings of First Alloy Workshop*, pages 71–80, 2006. 100

[GMR05] Hans-Gerhard Groß, Nikolas Mayer, and Javier Paredes Riano. *Assessing real-time component contracts through built-in evolutionary testing*, volume 3776 of *LNCS*, pages 107–122. Springer, Berlin, ALLEMAGNE, nov 2005. 44

[GPA+04a] Steffen Göbel, Christoph Pohl, Ronald Aigner, Martin Pohlack, Simone Röttger, and Steffen Zschaler. The comquad component container architecture. In *WICSA*, pages 315–320. IEEE Computer Society, 2004. 55

[GPA+04b] S. Göbel, C. Pohl, R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. The COMQUAD Component Container Architecture and Contract Negotiation. Technical Report TUD-FI04-04, Technische Universität Dresden, April 2004. 65

[Gro05] Hans-Gerhard Groß. *Component-Based Software Testing with UML*. Springer, 2005. 44

[GSS04] D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 276–277, 2004. 55

[HDPT04] J.L. Hellerstein, Y. Diao, S. Parekh, and D.M. Tilbury. *Feedback control of computing systems*. Wiley Online Library, 2004. 55, 64, 120, 123

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral compositions in Object-Oriented Systems. In Norman Meyrowitz, editor, *OOPSLA/ECOOP'90*, pages 169–180, Ottawa, Canada, October 1990. 19

[HHS10] Arnaud Hubaux, Patrick Heymans, and Pierre-Yves Schobbens. Supporting multiple perspectives in feature-based configuration: Foundations. Technical Report P-CS-TR PPFD-000001, University of Namur PReCISE Research Centre, Faculty of Computer Science Namur Belgium, March 2010. 93

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008. ACM. 125

[HLM+09] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a Consolidation Manager for Clusters. In *Proceedings of the ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'09)*, pages 41–50, New York, NY, USA, 2009. ACM. 123

[HMSW03] S. Hissam, G. Moreno, J. Stafford, and K. Wallnau. Enabling Predictable Assembly. *Journal of Systems and Software*, 65(3), 2003. 79

[HMW01] D. Hamlet, D. Mason, and D. Woit. Theory of Software Reliability Based on Components. In *ICSE'2001*. IEEE Computer, 2001. 79

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. 3

[HSS+10] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010. 103, 108

[HST+08] P. Heymans, P.-Y. Schobbens, J.-C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *Software, IET*, 2(3):281–302, June 2008. 100

[HT08] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *12th International Software Product Line Conference (SPLC'08)*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society. 93

[HTM09] Herman Hartmann, Tim Trew, and Aart Matsinger. Supplier independent feature modelling. In *SPLC'09*, pages 191–200. IEEE, 2009. 93, 110

[IBM01] IBM Research. Autonomic computing, 2001. http://www.research.ibm.com/autonomic/. 4, 55, 62, 120

[JDP03] Jean-Marc Jézéquel, Olivier Defour, and Noël Plouzeau. An mda approach to tame component based software development. In *Formal Methods for Components and Objects: Second International Symposium, FMCO 2003*, pages 260–275, 2003. 22

[JDT01] Jean-Marc Jézéquel, Daniel Deveaux, and Yves Le Traon. Reliable objects: Lightweight testing for oo languages. *IEEE Software*, 18:76–83, 2001. 3

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988. 1, 21

[JLSU87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, 1987. 80

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997. 21

[JYDZ09] Navendu Jain, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. Self-tuning, bandwidth-aware monitoring for dynamic data streams. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society. 81

[KAB+06] J. Kofron, J. Adamek, T. Bures, P. Jesek, V. Mencl, P. Parizek, and F. Plasil. Checking fractal component behaviour using behaviour protocols. In *Fractal Workshop, ECOOP 2006*, 2006. 28

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, May 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9. 125

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003. 2, 4, 5, 55, 88, 120, 123

[KC11] Filip Křikava and Philippe Collet. A Reflective Model for Architecting Feedback Control Systems. In *23rd International Conference on Software Engineering & Knowledge Engineering(SEKE'2011)*, , page 7, Miami Beach, USA, July 2011. 8, 123

[KCBF11] Filip Křikava, Philippe Collet, and Mireille Blay-Fornarino. Uniform and Model-Driven Engineering of Feedback Control Systems. In *8th IEEE/ACM International Conference on Autonomic Computing(ICAC'2011), short paper: To appear*, , page 2, Karlsruhe, Germany, June 2011. ACM. 123

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA). Technical Report CMU/SEI-90-TR-21, SEI, November 1990. 3, 6

[KKL+98] Kyo Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998. 91, 92, 93

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. 2, 26

[KPGV03a] Gail Kaiser, Janak Parekh, Philip Gross, and Giuseppe Valetto. Kinesthetics eXtreme : An External Infrastructure for Monitoring Distributed Legacy Systems. In *Proceedings of the autonomic computing workshop, 5. international workshop on active middleware services*, pages 4–13, Seattle, 2003. 55

[KPGV03b] Gall Kaiser, Janak Parekh, Phillip Gross, and Giuseppe Valetto. Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems. In *Proceedings of the autonomic computing workshop, fifth international workshop on active middleware services (AMS'03)*, pages 22–30. ACM, June 2003. 123

[KS98] J. Koistinen and A. Seetharaman. Worth-based multi-category quality-of-service negotiation in distributed object infrastructures. Technical report, HP Labs, 1998. 65

[KTS+09] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *31st International Conference on Software Engineering (ICSE'09), Tool demonstration*, pages 611–614, 2009. 102, 103, 104, 108

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley , April 2003. 2

[La95] D. C. Luckham and all. Specification and analysis of system architecture using rapide. *IEEE Trans. on Soft. Eng.*, 24(4):336–355, April 1995. 20

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994. 35

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999. 11, 19, 21

[LD10] Bao Le Duc. *A QoI-aware Framework for Adaptive Monitoring*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Paris, France, December 2010. 8, 81

[LDCMR10] Bao Le Duc, Philippe Collet, Jacques Malenfant, and Nicolas Rivierre. A QoI-aware Framework for Adaptive Monitoring. In *The Second International Conference on Adaptive and Self-adaptive Systems and Applications(ADAPTIVE 2010)*, , page 10, Lisbon, Portugal, November 2010. IARIA, Xpert Publishing Services. xii, 8, 80, 81, 85

[LPH04a] Hua Liu, Manish Parashar, and Salim Hariri. A component based programming framework for autonomic applications. In *Proc. of 1st International Conference on Autonomic Computing*, volume 9984357, pages 10–17. Citeseer, 2004. 55

[LPH04b] Hua Liu, Manish Parashar, and Salim Hariri. A component-based programming model for autonomic applications. *Autonomic Computing, International Conference on*, 0:10–17, 2004. 123

[LQS05] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM: A Component Framework for Constructing Resource-Aware, Configurable Middleware. *IEEE Distributed Systems Online*, 6(9), 2005. 69

[LRS00] Robert Laddaga, Paul Robertson, and Howard E. Shrobe. Results of the first international workshop on self adaptive software. In *Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers*, volume 1936 of *Lecture Notes in Computer Science*, pages 242–247. Springer, 2000. 2, 4, 54

[LS04] O. Loques and A. Sztajnberg. Customizing Component-Based Architectures by Contract. In *Proceedings of Component Deployment (CD 2004), Edinburgh, UK*, May 2004. 20, 55, 65

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33:709–724, October 2007. 2, 10, 44

[MAC+10] Olga Melekhova, Mohammed-Amine Abchir, Pierre Châtel, Jacques Malenfant, Isis Truck, and Anna Pappa. Self-adaptation in geotracking applications: Challenges, opportunities and models. In *The Second International Conference on Adaptive and Self-adaptive Systems and Applications(ADAPTIVE 2010)*, , pages 68–77, Lisbon, Portugal, November 2010. IARIA, Xpert Publishing Services. 123

[Mad03] Per Madsen. Unit testing using design by contract and equivalence partitions. In *XP*, pages 425–426, 2003. 44

[Mag99] Jeff Magee. Behavioral analysis of software architectures using ltsa. In *ICSE*, pages 634–637, 1999. 20

[MBC09] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.p.l.o.t.: software product lines online tools. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 761–762, New York, NY, USA, 2009. ACM. 102, 103, 104, 108, 109

[MBFCL08] Sébastien Mosser, Mireille Blay-Fornarino, Philippe Collet, and Philippe Lahire. Vers l'intégration dynamique de contrats dans des architectures orientées services : une experience applicative du modèle au code. In *2ème Conférence sur les Architectures Logicielles(CAL'08)*, , pages 63–77, Montréal, March 2008. C'epadu'es-'Editions. 35, 37

[MBFR08] Sébastien Mosser, Mireille Blay-Fornarino, and Michel Riveill. Web services orchestrations evolution: A merge process for behavioral evolution. In Ronald Morrison, Dharini Balasubramaniam, and Katrina E. Falkner, editors, *ECSA*, volume 5292 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2008. 37

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, 2009. 124

[MC10] Marcilio Mendonca and Donald Cowan. Decision-making coordination and efficient reasoning techniques for feature-based configuration. *Science of Computer Programming*, 75(5):311 – 332, 2010. Coordination Models, Languages and Applications (SAC'08). 93

[MCHB11] Raphaël Michel, Andreas Classen, Arnaud Hubaux, and Quentin Boucher. A formal semantics for feature cardinalities in feature diagrams. In Patrick Heymans, Krzysztof Czarnecki, and Ulrich W. Eisenecker, editors, *VaMoS*, ACM International Conference Proceedings Series, pages 82–89. ACM, 2011. 103, 121

[McI68] D. McIlroy. Mass-produced software components. In *First International Conference on Software Engineering*. NATO, August 1968. 2, 10

[MCLL07] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM (1)*, volume 4362 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007. 120

[Mer02] Annabelle Mercier. Etude bibliographique d?un modèle général de contractualisation pour les composants logiciels. Master's thesis, Université de Nice Sophia Antipolis, Nice, France, June 2002. 8

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988. 1

[Mey92] Bertrand Meyer. Applying "Design by contract". *IEEE Computer*, 25(10):40–51, October 1992. 2, 3, 13, 16, 44

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005. 103

[MJW08] Mohammad Ahmad Munawar, Michael Jiang, and Paul A. S. Ward. Monitoring multi-tier clustered systems with invariant metric relationships. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 73–80, New York, USA, 2008. ACM. 89

[MLM+06] Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter Brown, and Rebekah Metz. Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS, February 2006. 22, 36

[MM09] Krzysztof Czarnecki Marcilio Mendonca, Andrzej Wąsowski. Sat-based analysis of feature models is easy. In *SPLC'09*, pages 231–241. IEEE, 2009. 93, 100

[MMR+10] Rémi Mélisson, Philippe Merle, Daniel Romero, Romain Rouvoy, and Lionel Seinturier. Reconfigurable run-time support for distributed service component architectures. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 171–172. ACM, 2010. 113

[MPH+07] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE International Conference on Requirements Engineering (RE '07)*, pages 243–253, 2007. 93, 115, 116

[MRD08] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 815–824, New York, USA, 2008. ACM. 80

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Soft. Eng.*, 26(1), 2000. 10

[MTD08] Atif Manzoor, Hong-Linh Truong, and Schahram Dustdar. On the evaluation of quality of context. In *EuroSSC '08: Proceedings of the 3rd European Conference on Smart Sensing and Context*, pages 140–153, Berlin, Heidelberg, 2008. Springer-Verlag. 89

[MVL+08] Brice Morin, Gilles Vanwormhoudt, Philippe Lahire, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Managing variability complexity in aspect-oriented modeling. *Model Driven Engineering Languages and Systems*, pages 797–812, 2008. 6

[MW06] Mohammad Ahmad Munawar and Paul A. S. Ward. Adaptive monitoring in enterprise software systems. In *SIGMETRICS 2006 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2006. 81, 89

[MWCC08] Marcilio Mendonca, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *GPCE '08*, pages 13–22. ACM, 2008. 100, 109

[N. 01] N. Wang and D. Schmidt and K. Parameswaran and M. Kircher. Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *IEEE Distributed Systems Online special issue on Reflective Middleware*, 2(5), 2001. 55, 79

[NFG+06] L Northrop, P Feiler, R P Gabriel, J Goodenough, R Linger, T Longstaff, R Kazman, M Klein, D Schmidt, K Sullivan, and et al. Ultra-large-scale systems - the software challenge of the future. *Technical report Software Engineering Institute Carnegie Mellon University ISBN*, 0, 2006. 1, 4, 54

[Nor02] Linda M. Northrop. Sei's software product line tenets. *IEEE Softw.*, 19:32–40, July 2002. 5

[NRS10] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. Modelling feedback control loops for self-adaptive systems. *ECEASST*, 28, 2010. 123

[OMG97] Inc Object Management Group. Object Constraint Language Specification. Technical Report version 1.1, ad/97-08-08, IBM `www.software.ibm.com/ad/ocl`, September 1997. 13, 20

[Ope07] Open SOA. *SCA Service Component Architecture - Assembly Model Specification*, March 2007. Version 1.00. 22, 35, 123

[Oza07] Alain Ozanne. *Interact : un modèle général de contrat pour la garantie des assemblages de composants et services*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Paris, France, November 2007. 8

[Pah01] Claus Pahl. Components, Contracts and Connectors for the Unified Modelling Language UML. In Springer Verlag, editor, *FME2001 - Formal Methods Europe*, volume 2021 of *LNCS*, pages 259–277, 2001. 12, 20

[Pap03] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, pages 3–12. IEEE Computer Society, 2003. 36

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972. 2

[Par76] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1):1–9, 1976. 5

[PBCD11] Carlos Andres Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying design and runtime software adaptation using aspect models. *Sci. Comput. Program.*, 76(12):1247–1260, 2011. 124

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 5, 91, 93, 115

[PCBD10] C. A. Parra, A. Cleve, X. Blanc, and L. Duchien. Feature-based composition of software architectures. In *ECSA'10*, volume 6285 of *LNCS*, pages 230–245. Springer, 2010. 125

[PGS+07] Vahe Poladian, David Garlan, Mary Shaw, M. Satyanarayanan, Bradley Schmerl, and Joao Sousa. Leveraging resource prediction for anticipatory dynamic configuration. In *SASO '07: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems*, pages 214–223, Washington, DC, USA, 2007. IEEE Computer Society. 89

[Pin99] B. Joseph Pine. *Mass Customization: The New Frontier in Business Competition*. Harvard Business School Press, 1999. 5

[PKGJ08a] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *12th Software Product Line Conference*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society. 125

[PKGJ08b] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC'08*, pages 339–348. IEEE, 2008. 6

[PLS+00] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier. Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In *Proc. of ISORC 2000, 3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, Newport Beach, CA, March 15-17 2000. 65

[Plö02] Reinhold Plösch. Evaluation of Assertion Support for the Java Programming Language. In *Journal of Object Technology, Special issue: TOOLS USA 2002 proceedings*, volume 1,3, pages 5–17, 2002. 19

[PPK06] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model checking of software components: Combining java pathfinder and behavior protocol model checker. In *SEW*, pages 133–141. IEEE Computer Society, 2006. 21

[PS06] E. Putrycz and M. Slavescu. Solving Performance Issues in COTS-Based Systems. In *ICCBSS'2006*. IEEE Computer Society, 2006. 79

[PSGS04] Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic configuration of resource-aware services. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 604–613, Washington, DC, USA, 2004. IEEE Computer Society. 82, 89

[PSK+10] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*, pages 459–468. IEEE Computer Society, 2010. 125

[pur06] pure::variants. http://www.pure-systems.com/pure_variants.49.0.html, 2006. 102

[PV02] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Trans. on Soft. Eng.*, 28, November 2002. 11, 20, 21, 28

[PVL+10] Gilles Perrouin, Gilles Vanwormhoudt, Philippe Lahire, Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. Weaving Variability into Domain Metamodels. *Software and Systems Modeling Special issue*, page 22, 2010. 6, 125

[Rau00] Andreas Rausch. Software evolution in componentware using requirements/assurances contracts. In *ICSE*, pages 147–156, 2000. 22

[RBUW03] T. Ritter, M. Born, T. Unterschutz, and T. Weis. A QoS Metamodel and its Realization in a CORBA Component Infrastructure. In *Proceedings of the 36' HICSS Hawaii*, January 2003. 65

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. 68

[RRS+09] Daniel Romero, Romain Rouvoy, Lionel Seinturier, Sophie Chabridon, Denis Conan, and Nicolas Pessemier. *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*, chapter Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments, pages 113–135. Chapman and Hall/CRC, July 2009. 123

[RSP03a] R. Reussner, H. Schmidt, and I. Poernomo. Reliability Prediction for Component-based Software Architectures. *Journal of Systems and Software*, 66(3), 2003. 79

[RSP03b] Ralf H. Reussner, Heinz W. Schmidt, and Iman H. Poernomo. Reasoning on software architectures with contractually specified components. In A. Cechich, M. Piattini, and A. Vallecillo, editors, *Component-Based Software Quality: Methods and Techniques*. Springer-Verlag, Berlin, Germany, 2003. 22

[RW07] Mark-Oliver Reiser and Matthias Weber. Multi-level feature trees: A pragmatic approach to managing highly complex product families. *Requir. Eng.*, 12(2):57–75, 2007. 93, 110

[SAD+02] A. Sassen, G. Amoros, P. Donth, K. Geihs, J. J?z?quel, K. Odent, N. Plouzeau, and T. Weis. Qccs: A methodology for the development of contract-aware components based on aspect-oriented design. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Enschede, The NetherLands, mar 2002. 22

[SBD08] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using Components for Architecture-Based Management: The Self-Repair Case. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. ACM, May 2008. 123

[SBRCT08] S. Segura, D. Benavides, A. Ruiz-Cortes, and P. Trinidad. Automated merging of feature models using graph transformations. *Post-proceedings of the Second Summer School on GTTSE*, 5235:489–505, 2008. 100

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006. 2

[SHTB07] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007. 92, 100

[SLB+11] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE'11*, 2011. to appear. 93, 99, 100, 125

[SLM98] D.C. Schmidt, D.L. Levine, and S. Mungee. The Design of the TAO Real-time Object Request Broker. *Computer Communications*, 21(4), 1998. 65

[Smi80] R. G. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, 29:1104–1113, 1980. 57, 65

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A component model engineered with components and aspects. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *CBSE*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2006. 37

[SPHM09] Germain Saval, Jorge Pinna Puissant, Patrick Heymans, and Tom Mens. Some challenges of feature-based merging of class diagrams. In David Benavides, Andreas Metzger, and Ulrich W. Eisenecker, editors, *VaMoS*, volume 29 of *ICB Research Report*, pages 127–136. Universität Duisburg-Essen, 2009. 6, 126

[SPTU05] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, 2005. 4

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009. 2, 4, 123

[sta07] SCA standard. http://www.osoa.org/, 2007. 36, 114

[SZLT+10] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and Gøran K. Olsen. Developing a software product line for train control: A case study of cvl. In Jan Bosch and Jaejoon Lee, editors, *SPLC*, volume 6287 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2010. 125

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition edition, 2002. 2, 3, 10, 44

[TBC+09] Thein Than Tun, Quentin Boucher, Andreas Classen, Arnaud Hubaux, and Patrick Heymans. Relating requirements and feature configurations: A systematic approach. In *SPLC'09*, pages 201–210. IEEE, 2009. 93

[TBD+04] Hanh-Middi Tran, Philippe Bedu, Laurence Duchien, Hai-Quan Nguyen, and Jean Perrin. Toward structural and behavioral analysis for component models. In *SAVBCS 2004, 12th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, NewPort Beach, California, USA, November 2004. 22

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE'09*, pages 254–264. IEEE, 2009. 94, 95, 98, 102, 106, 108

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007. 89

[TDJ99] Yves Le Traon, Daniel Deveaux, and Jean-Marc Jézéquel. Self-testable components: From pragmatic tests to design-for-testability methodology. In *TOOLS (29)*, pages 96–107. IEEE Computer Society, 1999. 44

[TLR+09] Jean-Yves Tigli, Stephane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. Wcomp middleware for ubiquitous computing: Aspects and composite event-based web services. *Annales des Télécommunications*, 64(3-4):197–214, 2009. 38

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE'99*, pages 107–119. ACM, 1999. 2, 93

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance. *Journal of Software Maintenance*, 10:75–92, March 1998. 103

[vdS04] Tijs van der Storm. *Variability and Component Composition*, pages 157–166. Software Reuse: Methods, Techniques and Tools, 2004. 110

[VFH05] Egon Valentini, Gerhard Fliess, and Edmund Haselwanter. A framework for efficient contract-based testing of software components. In *Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume Vol 2, pages 219–222. IEEE Computer Society, 2005. 44

[VG07] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 233–242, Washington, DC, USA, 2007. IEEE Computer Society. 6, 103

[vGN10] P. M. van den Broek, I. Galvao Lourenco da Silva, and J. A. R. Noppen. Merging feature models. In *14th International Software Product Line Conference, Volume 2, Jeju Island, South Korea*, pages 83–89, Lancaster, UK, 2010. Lancaster University, Lancaster, UK. 101

[vO02] Rob van Ommering. Building product populations with software components. In *ICSE '02*, pages 255–265. ACM, 2002. 110

[WBGP01] Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau. A UML Meta-model for Contract Aware Components. In *UML 2001 - The Unified Modeling Language*, volume 2185 of *Lecture Notes in Computer Science*, pages 442–456. Springer Verlag, October 2001. 12, 13, 20

[WCR09] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC'09*, volume 446 of *ICPS*, pages 211–220. ACM, 2009. 125

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. 98

[WMD08] Guillaume Waignier, Anne-Françoise Le Meur, and Laurence Duchien. Architectural specification and static analyses of contractual application properties. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *QoSA*, volume 5281 of *Lecture Notes in Computer Science*, pages 152–170. Springer, 2008. 120, 122

[WSHK01] K. Wallnau, J. Stafford, S. Hissam, and M. Klein. On the Relationship of Software Architecture to Software Component Technology. In *ECOOP'2001 WCOP Workshop*, 2001. 79

[WSMD08] Guillaume Waignier, Prawee Sriplakich, Anne-Françoise Le Meur, and Laurence Duchien. A model-based framework for statically and dynamically checking component interactions. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *LNCS*, pages 371–385. Springer, 2008. 120, 122

[Xte11] Xtext. `http://www.eclipse.org/Xtext/`, 2009/2011. 108

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM. 121, 123, 124

[ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. *Product Line Engineering with the UML: Deriving Products*, chapter 15, pages 557–586. Number 978-3-540-33252-7 in Software Product Lines: Reasearch Issues in Engineering and Management. Springer Verlag, 2006. 6, 103